IF107 - Logique et preuve de programmes

Frédéric Herbreteau
ENSEIRB-MATMECA (Bordeaux INP)

2018/2019

Table des matières

1 Introduction **5**

I Fondements du raisonnement mathématique **7**

2 Structures inductives - Preuve par induction **9**
2.1 Raisonnement par récurrence sur $\mathbb{N}$ 9
2.1.1 Premier principe d'induction 9
2.1.2 Second principe d'induction 10
2.2 Définitions inductives et preuves par induction structurelle 11
2.2.1 Ensembles définis inductivement 11
2.2.2 Fonctions définies inductivement 12
2.2.3 Preuves par induction 13
2.2.4 Types Abstraits de Données, Programmation fonctionnelle et induction 14

3 Logique propositionnelle et preuve **17**
3.1 Introduction : pourquoi la logique ? 17
3.2 Préliminaires sur le raisonnement mathématique 18
3.2.1 Quelques propriétés de l'implication 19
3.2.2 Confusions fréquentes à éviter 19
3.3 Logique propositionnelle 19
3.3.1 Syntaxe 19
3.3.2 Sémantique 20
3.4 Système de déduction, preuve 21
3.4.1 Représentation du raisonnement : les séquents 22
3.4.2 Démonstrations logiques 23
3.5 Connecteurs logiques supplémentaires 25
3.6 Formes normales 26
3.6.1 Forme normale négative 27
3.6.2 Forme normale conjonctive 27
3.6.3 Forme normale disjonctive 28

4 Logique des prédicats 31
4.1 Syntaxe . . . 31
4.1.1 Langage . . . 31
4.1.2 Formules . . . 33
4.2 Sémantique . . . 34
4.2.1 Structure : interprétation d'un langage . . . 34
4.2.2 Valuation . . . 35
4.2.3 Sémantique des formules . . . 35
4.3 Substitution . . . 37
4.4 Système de déduction, preuve . . . 37

II Preuve de programmes 39

5 Motivation pour la preuve de programmes 41

6 Spécification et sémantique des programmes 43
6.1 Spécification . . . 43
6.2 Programmes et sémantique . . . 44
6.2.1 While programs . . . 44
6.2.2 Sémantique, correction et terminaison . . . 44

7 Preuve de programmes sans boucles 47
7.1 Terminaison . . . 47
7.2 Correction . . . 47
7.2.1 Calcul de Hoare . . . 47

8 Preuve de programmes avec boucles 51
8.1 Terminaison : variants . . . 51
8.1.1 Ensembles bien fondés . . . 51
8.1.2 Suites décroissantes d'environnements . . . 52
8.1.3 Limites de l'approche . . . 53
8.2 Correction : invariants de boucle . . . 53
8.2.1 Extension du calcul de Hoare pour `while` . . . 53
8.2.2 Comment trouver des invariants de boucle ? . . . 55
8.2.3 Intégration de la preuve de terminaison . . . 57

Chapitre 1

Introduction

Ce cours constitue une introduction au raisonnement formel et à la preuve de programme. La première partie aborde l'induction, la logique et la preuve. En particulier, nous montrons comment la preuve formelle permet de raisonner à un niveau purement syntaxique, tout en garantissant la validité des théorèmes prouvés.

La seconde partie est consacrée à la preuve de programmes. La logique y est vue comme un outil de spécification, et comme un outil de raisonnement (preuve). La logique de Hoare est introduite pour la preuve des programmes. Les ensembles bien fondés sont utilisés pour la preuve de terminaison.

Objectifs principaux :
- Induction : définition, algorithmes et preuve sur des collections infinies d'objets finis
- Logique : formalisation, raisonnement, preuve
- Preuve de programmes : spécification, annotation, correction, terminaison.

Les références bibliographiques principales pour ce cours sont :
- A. Arnold et I. Guessarian, *Mathématiques pour l'informatique*, Masson, 1997.
- J. Stern, *Fondements mathématiques de l'informatique*, éditions Odile Jacob.
- R.W. Floyd, *Assigning Meanings to Programs*. Proc. Amer. Math. Soc. Symposia in Applied Mathematics, 1967.
- C.A.R. Hoare, *An Axiomatic Basis for Computer Programming*. Communications of Computer Science, 1969.
- E.W. Dijkstra, *Guarded commands, nondeterminacy and formal derivation of program*. Communications of the ACM, 18(8) :453–457, August 1975.
- Christine Paulin-Mohring, *Éléments de logique pour l'informatique*, `https://www.lri.fr/~paulin/Logique/`, 2018

Ceci est une version préliminaire du polycopié. Merci de signaler toute erreur.

Première partie

Fondements du raisonnement mathématique

Chapitre 2

Structures inductives - Preuve par induction

Les définitions inductives permettent de construire des objets finis à partir d'autres objets, en suivant certaines règles bien précises. Elles permettent également d'appréhender des objets infinis, représentés par une succession d'approximations finies. Elles interviennent notamment dans la description des structures de données (listes, arbres, etc).

En informatique, nous ne pouvons manipuler que des objets finis. Les définitions inductives représentent de façon finie des collections infinies d'objets finis. Les démonstrations par récurrence permettent de raisonner sur ces objets.

2.1 Raisonnement par récurrence sur $\mathbb{N}$

2.1.1 Premier principe d'induction

Théorème 2.1 Soit $P(n)$ un prédicat (ou propriété) dépendant de l'entier n. Si les deux conditions sont vérifiées :

(B) $P(0)$ est vrai,

(I) $\forall n \in \mathbb{N},\ (P(n)\ \Rightarrow\ P(n+1))$,

alors $\forall n \in \mathbb{N},\ P(n)$ est vrai. ♦

(B) est l'étape de base, (I) est l'étape d'induction. On peut remplacer 0 par tout entier n_0.

Preuve. Par l'absurde. On suppose (B) et (I) mais $\exists n.\ P(n) = faux$. Soit :

$$X = \{k \in \mathbb{N} \mid P(k) \text{ est faux}\}$$

Si X est non vide, il admet un plus petit élément n. D'après la condition (B), $n \neq 0$. Donc $n-1$ est un entier et $n-1 \notin X$, c'est à dire $P(n-1)$ est vrai. Par (I) nous avons alors $P(n)$ est vrai ce qui contredit $n \in X$. ♦

Attention, (I) n'affirme pas que $P(n+1)$ ou $P(n)$ soit vrai, mais seulement que si $P(n)$ est vrai, alors $P(n+1)$ est vrai.

La preuve est faite lorsque (I) *et* (B) ont été prouvées.

Exemple 2.2
On veut calculer la somme $S_n = 1 + 2 + \cdots + n$. On remarque $2S_1 = 2 = 1 \times 2$, $2S_2 = 2 + 4 = 2 \times 3$, $2S_3 = 2 + 4 + 6 = 3 \times 4$. On conjecture que pour tout $n > 0$, $2S_n = n(n+1)$.

On le montre par récurrence :

— (B) $2S_1 = 1 \times 2$ donc $P(1)$ est vrai

— (I) $2S_{n+1} = 2S_n + 2(n+1) = \mathbf{n(n+1)} + 2(n+1) = (n+1)(n+2)$ donc sous hypothèse que $P(n)$ est vrai, $P(n+1)$ est vrai.

Donc $\forall n \geq 1$, $P(n)$ est vrai. ♦

2.1.2 Second principe d'induction

Parfois, pour prouver $P(n)$ vrai, il est nécessaire d'utiliser explicitement plus que $P(n-1)$ vrai, c'est à dire que P est vraie pour $0, 1, \ldots, n-1$.

Théorème 2.3 Soit $P(n)$ un prédicat dépendant de l'entier n. Si la proposition suivante est vérifiée :

$$(\mathbf{I_2}) \qquad \forall n \in \mathbb{N},\ ((\forall k < n,\ P(k)) \ \Rightarrow\ P(n))$$

alors $\forall n \in \mathbb{N}$, $P(n)$ est vraie. ♦

Notons que l'hypothèse de base est dissimulée dans le cas $n = 0$:

$$(\forall k < 0,\ P(k)) \ \Rightarrow\ P(0)$$

comme $(\forall k < 0, P(k))$ est toujours vraie, $P(0)$ doit également être vraie.

Sur $\mathbb{N}$, les deux principes d'induction sont équivalents, mais seul le second principe se généralise à des ensembles ordonnés plus généraux.

Exemple 2.4
on veut démontrer que tout entier $n \geq 2$ est décomposable en un produit de nombres premiers. Notons $P(n)$ la propriété « n est décomposable en un produit de nombres premiers ». Soit $n \geq 2$, et supposons $\forall k \in \{2, \ldots, n-1\}, P(k)$ (induction). Nous avons alors :

— soit n est premier, alors il est évidemment décomposable en un produit d'un nombre premier ;

— soit n n'est pas premier, alors il existe $a, b \in \{2, \ldots, n-1\}$ tels que $n = ab$. Par induction, $P(a)$ et $P(b)$ sont vraies, et donc n est décomposable en le produit des décompositions de a et de b.

♦

2.2 Définitions inductives et preuves par induction structurelle

En informatique, de nombreuses structures de données et de nombreuses fonctions sont définies de manière inductive. Cela permet de spécifier des données de taille non bornée, ou des fonctions à récursion non bornée *a priori* à partir d'une définition finie, donc manipulable.

2.2.1 Ensembles définis inductivement

Définition 2.5 Soit E un ensemble. Une définition inductive d'une partie X de E consiste en la donnée :
— d'un sous-ensemble B de E ;
— d'un ensemble K d'opérations $f : E^{a(f)} \to E$ où $a(f) \in \mathbb{N}$ est l'arité de f.
X est défini comme étant le plus petit ensemble vérifiant les assertions (B) et (I) suivantes :

(B) $B \subseteq X$;

(I) $\forall f \in K,\ \forall x_1, \ldots, x_{a(f)} \in X,\ f(x_1, \ldots, x_{a(f)}) \in X$.

Exemple 2.6
— La partie X de $\mathbb{N}$ définie inductivement par :

(B) $0 \in X$;

(I) $n \in X \ \Rightarrow s(n) \in X$ ($s(n)$ est noté $n + 1$ usuellement)

est $\mathbb{N}$ tout entier.

— La partie X de Σ^* définie inductivement par :

(B) $\varepsilon \in X$;

(I) $u \in X \ \Rightarrow\ \forall a \in \Sigma, ua \in X$

est Σ^* lui-même.

— En informatique, les définitions sont bien souvent inductives. L'ensemble des expressions défini par :

(B) $\epsilon \in X$, $\emptyset \in X$ et $a, b \in X$

(I) $u, v \in X \ \Rightarrow\ uv \in X, u + v \in X$ et $u^\star \in X$

est l'ensemble des expressions régulières sur l'alphabet $\{a, b\}$.

— L'ensemble AB des arbres binaires est défini par induction comme suit :

(B) $\emptyset \in AB$;

(I) $g, d \in AB \ \Rightarrow\ \forall a \in \Sigma,\ a(g, d) \in AB$

Pour un arbre binaire, l'ensemble K des opérations est l'ensemble a des symboles de Σ.

♦

Théorème 2.7 Si X est défini par les conditions (B) et (I), tout élément de X peut s'obtenir à partir de la base en appliquant un nombre fini d'étapes inductives. ♦

Preuve. Définissons la suite d'ensembles :

— $X_0 = B$;

— $X_{n+1} = X_n \cup \{f(x_1, \ldots, x_{a(f)}) \,|\, x_1, \ldots, x_{a(f)} \in X_n$ et $f \in K\}$

On montre par récurrence que $X_n \subseteq X$ pour tout $n \geq 0$. C'est à dire $X_\omega = \bigcup_{n\geq 0} X_n \subseteq X$.

Il faut maintenant montrer que X_ω vérifie (B) et (I). D'une part $B = X_0 \subseteq X_\omega$. D'autre part, soit $f \in K$ et $x_1, \ldots, x_{a(f)} \in X_\omega$. Chaque x_i appartient à un ensemble $X_{n_i} \in X_\omega$. Soit n le plus grand n_i. Alors, pour tout i, $x_i \in X_n$ (car $X_{n+1} = X_n \cup \ldots$) donc $f(x_1, \ldots, x_{a(f)}) \in X_{n+1}$. Puisque $X_{n+1} \subseteq X_\omega$, X_ω vérifie (I). ♦

2.2.2 Fonctions définies inductivement

Une définition inductive d'un ensemble X est *non-ambiguë* si tout élément x de X s'obtient de manière unique en appliquant les règles (I) et (B).

Exemple 2.8
La définition suivante de $\mathbb{N}^2$ est ambiguë :

(B) $(0, 0) \in \mathbb{N}^2$

(I1) $(n, m) \in \mathbb{N}^2 \Rightarrow (n + 1, m) \in \mathbb{N}^2$

(I2) $(n, m) \in \mathbb{N}^2 \Rightarrow (n, m + 1) \in \mathbb{N}^2$

En effet, $(1, 1) \in \mathbb{N}^2$ s'obtient en appliquant (I_1) puis (I_2) à partir de $(0, 0)$, mais également en appliquant d'abord (I_2) puis (I_1). ♦

Définition 2.9 Soit $X \subseteq E$ un ensemble défini inductivement de façon non-ambiguë, et soit F un ensemble quelconque. La définition inductive d'une application α de X dans F consiste en :

(B) la donnée de $\alpha(x) \in F$ pour tout élément $x \in B$;

(I) et pour chaque $f \in K$, l'expression de $\alpha(f(x_1, \ldots, x_{a(f)}))$ à partir des $x_1, \ldots, x_{a(f)}$ et des $\alpha(x_1), \ldots, \alpha(x_{a(f)})$.

♦

Remarque : si la définition de X est ambiguë, α n'est pas une application car elle peut avoir plusieurs images.

Exemple 2.10

— La fonction factorielle de $\mathbb{N}$ dans $\mathbb{N}$ est définie inductivement par :

  — $fact(0) = s(0)$ (ou $fact(0) = 1$)
  — $fact(s(n)) = s(n) \times fact(n)$ (ou $fact(n+1) = (n+1) \times fact(n)$)

— La hauteur d'un arbre binaire est la fonction de AB dans $\mathbb{N}$ définie inductivement par :
  — $h(\emptyset) = 0$
  — $h(a(g, d)) = \max(h(g), h(d)) + 1$

— Le nombre de nœuds dans un arbre binaire est la fonction de AB dans $\mathbb{N}$ inductivement définie par :
  — $n(\emptyset) = 0$
  — $n(a(g, d)) = 1 + n(g) + n(d)$

— Le parcours infixe (liste des étiquettes) d'un arbre binaire est la fonction inductive de AB dans Σ^* définie par :
  — $infixe(\emptyset) = \varepsilon$
  — $infixe(a(g, d)) = infixe(g).a.infixe(d)$

♦

2.2.3 Preuves par induction

Généralisation du principe de récurrence aux ensembles définis inductivement.

Théorème 2.11 Soit X un ensemble défini inductivement et soit $P(x)$ un prédicat exprimant une propriété de l'élément x de X. Si les conditions suivantes sont vérifiées :

(B″) $P(x)$ est vrai pour tout $x \in B$;

(I″) pour chaque $f \in K$, $(P(x_1) \wedge \cdots \wedge P(x_{a(f)})) \Rightarrow P(f(x_1, \ldots, x_{a(f)}))$

alors $P(x)$ est vrai pour tout $x \in X$. ♦

Vérifier (B'') et (I'') constitue une *preuve par induction* de P sur X.

Preuve. Soit Y l'ensemble des éléments x tels que $P(x)$ est vrai. On a d'une part $B \subseteq Y$, et Y vérifie les clauses inductives (I) de la définition de X, par (I''). Donc $X \subseteq Y$. ♦

Exemple 2.12
Montrons que le nombre de nœuds $n(x)$ d'un arbre binaire $x \in AB$ est tel que $n(x) \leq 2^{h(x)} - 1 \equiv P(x)$

— $P(\emptyset)$ est vrai puisque $n(\emptyset) = 0$ et $h(\emptyset) = 0$
— Supposons $P(g)$ et $P(d)$ où $g, d \in AB$. Alors, pour tout $a \in \Sigma$:

$$\begin{aligned} n(a(g,d)) &= 1 + n(g) + n(d) \\ &\leq 1 + 2^{h(g)} - 1 + 2^{h(d)} - 1 \\ &\leq 2^{\max(h(g),h(d))+1} - 1 \\ &\leq 2^{h(a(g,d))} - 1 \end{aligned}$$

Nous avons donc pour tout $a \in \Sigma$, $(P(g) \wedge P(d)) \Rightarrow P(a(g,d))$.
Nous avons montré (B'') et (I''), donc $P(x)$ est vrai pour tout $x \in AB$. ♦

2.2.4 Types Abstraits de Données, Programmation fonctionnelle et induction

Un Type Abstrait de Données (TAD) est généralement défini de manière inductive.

Exemple 2.13
L'ensemble des arbres binaires AB sur l'alphabet Σ est inductivement défini par :

(B) $\emptyset \in AB$

(I) pour tout $a \in \Sigma$ et $g, d \in AB$, $a(g,d) \in AB$

Dans de nombreux langages fonctionnels (*Caml, Haskell, etc), il est possible de définir des types inductifs :

```
type 'a ab =   Empty
             | Node of 'a * ('a ab) * ('a ab)
```

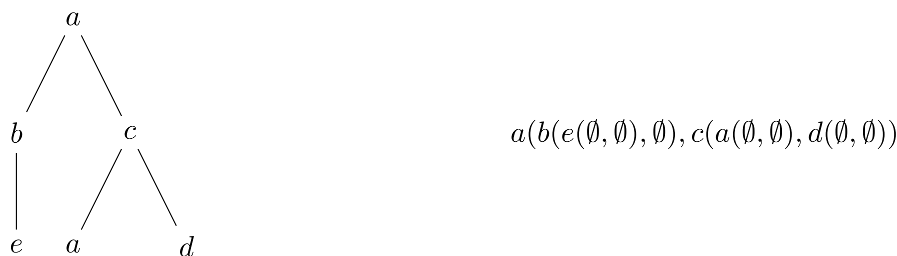
Ceci correspond à la définition d'un TAD avec deux constructeurs :

```
empty : tree
node  : elmt * tree * tree -> tree
```

♦

Cette définition introduit l'ensemble des *constructeurs* du TAD. Tout élément est obtenu par une application de ces constructeurs un nombre de fois fini.

Exemple 2.14
Représentation d'un arbre par un terme :



Le terme est construit par application successives des constructeurs du TAD.
De même, on peut représenter cet arbre en utilisant le type Caml introduit précédemment :

```
Node("a",
     Node("b",
          Node("e",
               Empty,
               Empty),
          Empty),
```

```
  Node("c",
       Node("a",
            Empty,
            Empty),
       Node("d",
            Empty,
            Empty)))
```

Les opérations du TAD sont définies récursivement en utilisant la structure inductive du type.

Exemple 2.15
Ensemble des étiquettes qui apparaissent dans l'arbre :

```
E : tree -> elmt set
```

$$\texttt{E}(\emptyset) = \emptyset$$
$$\texttt{E}(a(g,d)) = \{a\} \cup \texttt{E}(g) \cup \texttt{E}(d)$$

Appartenance d'une lettre à un arbre :

```
member : tree * elmt -> bool
```

$$\texttt{member}(\emptyset, a) = false$$
$$\texttt{member}(a(g,d), a) = true$$
$$\texttt{member}(b(g,d), a) = \texttt{member}(g,a) \text{ or } \texttt{member}(d,a) \qquad [b \neq a]$$

Les fonctions caml suivent le même principe :

```
let rec e t =
  match t with
    Empty -> []
  | Node(a, g, d) -> List.append (e g) (a :: (e d))

let rec member t a =
  match t with
    Empty -> false
  | Node(b, g, d) -> if (a = b) then
                       true
                     else
                       (member g a) || (member d a)
```

On peut alors prouver la correction des fonctions récursives ainsi définies.

Exemple 2.16
Preuve de correction de $\texttt{E}(x)$:

(Base) $\texttt{E}(\emptyset) = \emptyset$, ok

(Induction) On suppose la propriété vraie pour g et d, alors $\texttt{E}(a(g,d)) = \{a\} \cup \texttt{E}(g) \cup \texttt{E}(d)$ est bien l'ensemble des étiquettes de $a(g,d)$ par hypothèse d'induction.

Preuve de correction de $\texttt{member}(x,a)$. On va montrer la propriété suivante :

$$\texttt{member}(x,a) \Leftrightarrow a \in \texttt{E}(x)$$

(Base) $\texttt{member}(\emptyset, a) = false$ et $a \notin \texttt{E}(\emptyset) = \emptyset$, ok

(Induction) On distingue deux cas :

— $\texttt{member}(a(g,d),a) = true$ et $a \in \texttt{E}(a(g,d)) = \{a\} \cup \texttt{E}(g) \cup \texttt{E}(d)$.

— $\texttt{member}(b(g,d),a) = \texttt{member}(g,a)$ or $\texttt{member}(d,a)$. Par hypothèse d'induction, la propriété est vérifiée pour g et d. Donc :

$$\begin{array}{rl} & \texttt{member}(g,a) \Leftrightarrow a \in \texttt{E}(g) \\ \text{et} & \texttt{member}(d,a) \Leftrightarrow a \in \texttt{E}(d) \end{array}$$

Alors :

$$\begin{array}{lll} \texttt{member}(b(g,d),a) \Leftrightarrow a \in (\texttt{E}(g) \cup \texttt{E}(d)) & \\ \texttt{member}(b(g,d),a) \Leftrightarrow a \in (\{b\} \cup \texttt{E}(g) \cup \texttt{E}(d)) & [a \neq b] \\ \texttt{member}(b(g,d),a) \Leftrightarrow a \in \texttt{E}(b(g,d)) & \end{array}$$

On a ainsi (quasiment) prouvé le code Caml : ici le texte est suffisament proche de la définition inductive pour en être convaincu. ♦

Chapitre 3

Logique propositionnelle et preuve

3.1 Introduction : pourquoi la logique ?

Syllogismes (Aristote, -384 à -322) :

```
Tous les hommes sont mortels        Aucun chat [n']a huit queues
Socrate est un homme                Un chat a une queue de plus qu'aucun chat
----------------------------        -----------------------------------------
Socrate est mortel                  Un chat a neuf queues
```

Formaliser = associer une *sémantique* (mathématique) à la syntaxe qui permet de raisonner sur cette syntaxe.

Raisonnement des sophismes basé sur la règle du *Modus Ponens* :

$$\frac{A \Rightarrow B \qquad A}{B}$$

Ici, la syntaxe : `A` et `B` montrent une identité sur les objets utilisés dans le raisonnement.

Retour sur le premier sophisme. Modélisation :

```
homme(x):   vrai ssi x est un homme
mortel(x):  vrai ssi x est mortel
Socrate:    nom
```

Expression du raisonnement « type Modus Ponens » :

```
homme(x) -> mortel(x)
homme(Socrate)
---------------------
mortel(Socrate)
```

Pour le second sophisme, on interprète « aucun chat » comme l'ensemble de chats de cardinalité 0, et « un chat » comme l'ensemble de chats de cardinalité 1 :

```
nbq(k, n):    vrai ssi un ensemble de cardinalité k a n queues
```

Cette fois-ci, le raisonnement ne suit pas la règle du Modus Ponens (ni aucune règle ≪ raisonnable ≫) :

```
! nbq(1, 8)
nbq(0, n) -> nbq(1, n+1)
------------------------
nbq(1, 9)
```

La formalisation logique permet de lever l'ambiguïté de la langue naturelle, et par conséquent de raisonner sur des énoncés.

3.2 Préliminaires sur le raisonnement mathématique

Une *proposition* est une assertion qui est soit vraie, soit fausse. Par exemple, ≪ $p \implies p$ ≫ (p implique p) est une assertion qui est vraie alors que ≪ $2+2=5$ ≫ est une assertion qui est fausse.

Une *formule* exprime une propriété d'un objet ou une relation entre objets. Par exemple : ≪ $2+2=x$ ≫ est une formule. Une formule n'est vraie ou fausse que lorsqu'une valeur a été attribuée aux objets. Par exemple, si x vaut 4 alors la formule précédente est vraie. Elle est fausse pour toute autre valeur de x (en base 10).

Un *théorème* est une formule qui est toujours vraie.

Le raisonnement mathématique a pour but de *prouver des théorèmes*. Il se base sur la notion de *déduction* : en partant d'*hypothèses*, on déduit des *conclusions*, en appliquant une règle de *déduction*.

Ceci se note généralement :

$$p \implies q \qquad \text{ou} \qquad \frac{p}{q}$$

où p et q sont des des propositions. L'implication est un théorème si à chaque fois que p est vraie (l'hypothèse), q est également vraie (la conclusion).

Exemple 3.1
Les énoncés suivants sont des théorèmes bien connus de la théorie des ensembles :
— $A \cap B = C \implies C \subseteq A$ et $C \subseteq B$
— $A \cup B = C \implies A \subseteq C$ et $B \subseteq C$.

On note $\neg p$ la *négation* de la proposition p.

3.2.1 Quelques propriétés de l'implication

— L'implication est *transitive* : $p \implies q$ et $q \implies r$ impliquent $p \implies r$, c'est à dire :
$$[(p \implies q) \text{ et } (q \implies r)] \implies (p \implies r)$$

— L'implication $p \implies q$ et sa *contraposée* $(\neg q \implies \neg p)$ affirment la même chose.
C'est le principe même de la *preuve par contradiction* où on suppose p et $\neg q$ et on montre $\neg q \implies \neg p$.

— Ces 4 énoncés sont *équivalents* :

1. $p \implies q$ et $q \implies p$
2. $p \implies q$ et $\neg p \implies \neg q$
3. $p \Leftrightarrow q$
4. $\neg p \Leftrightarrow \neg q$

— Règles de raisonnement :
— *modus ponens* : $[p$ et $(p \implies q)] \implies q$
— *modus tollens* : $[\neg q$ et $(p \implies q)] \implies \neg p$

3.2.2 Confusions fréquentes à éviter

Attention à ne pas confondre la contraposée $\neg q \implies \neg p$ de l'implication $p \implies q$ avec l'*implication réciproque* $q \implies p$ qui exprime, généralement, autre chose. Voir exemple 3.1.

Attention également : si $p \implies q$ est fausse, la réciproque $q \implies p$ n'est pas nécessairement vraie. On peut faire l'analogie avec $B \not\subseteq A$ qui n'implique pas nécessairement $A \subseteq B$.

3.3 Logique propositionnelle

3.3.1 Syntaxe

Soit $P = \{p, p', q, q', \ldots\}$ un ensemble de symboles propositionnels.

Définition 3.2 Une formule de la logique propositionnelle est une suite de symboles dans $P \cup \{\implies, \neg, (,)\}$ telle que :

1. tout symbole de P est une formule
2. si ϕ est une formule, alors $\neg\phi$ et (ϕ) sont également des formules
3. si ϕ et ψ sont des formules, alors $\phi \implies \psi$ est une formule

Toute formule s'obtient par la répétition des étapes ci-dessus un nombre fini de fois.

Exemple 3.3
Soit $P = \{p, q, r\}$ un ensemble de variables propositionnelles. p, $p \implies q$, $\neg p$, $\neg(p \implies q)$, $\neg p \implies q$ et $p \implies (\neg q \implies r)$ sont des formules de la logique propositionnelle. ♦

Une *substitution (syntaxique)* consiste à remplacer dans une formule un symbole par une autre formule.

Définition 3.4 Une substitution est une application σ de P dans l'ensemble des formules, telle que $\sigma(\phi)$ est définie pour toute formule ϕ par :

1. si ϕ est une proposition p, alors $\sigma(\phi) = \sigma(p)$
2. si $\phi = \neg\phi'$ alors $\sigma(\phi) = \neg\sigma(\phi')$
3. si $\phi = (\psi \implies \psi')$ alors $\sigma(\phi) = (\sigma(\psi) \implies \sigma(\psi'))$

♦

Exemple 3.5
Soit la substitution σ telle $\sigma(p) = q$ et $\sigma(q) = (p \implies q)$, alors $\sigma(p \implies q) = (q \implies (p \implies q))$. ♦

3.3.2 Sémantique

L'algèbre de Boole $\mathbb{B} = \{0, 1\}$ munie des opérateurs $+$, $\cdot$ et $_$ a les propriétés suivantes :

— $+$ et $\cdot$ sont idempotentes, associatives, commutatives et distributives l'une par rapport à l'autre,
— $x \cdot (x + y) = x = (y \cdot x) + x$
— $x \cdot 0 = 0$, $x + 0 = x$, $x \cdot 1 = x$ et $x + 1 = 1$
— $x \cdot \overline{x} = 0$ et $x + \overline{x} = 1$

Une *interprétation* est une application de P dans l'algèbre de Boole. Par exemple, l'application I définie par $I(p) = 0$ et $I(q) = 1$ est une interprétation sur $P = \{p, q\}$.

Étant donnée une interprétation I, on définit la *valeur de vérité*, $[\phi]_I$, d'une formule ϕ par :

— si ϕ est une proposition p, alors $[\phi]_I = I(p)$
— si $\phi = \neg\phi'$ alors $[\phi]_I = \overline{[\phi']_I}$
— si $\phi = (\psi \implies \psi')$ alors $[\phi]_I = \overline{[\psi]_I} + [\psi']_I$

Si $[\phi]_I$ vaut 1, on dit que ϕ est *vraie* (pour I).

Alternativement, la sémantique des connecteurs logiques peut également être donnée sous la forme d'une *table de vérité* :

$\neg$	
0	1
1	0

$\implies$	0	1
0	1	1
1	0	1

Définition 3.6 Une formule ϕ est :

- *valide* si $[\phi]_I = 1$ pour tout interprétation I (on dit également que ϕ est une *tautologie*)
- *satisfaisable* s'il existe I telle que $[\phi]_I = 1$
- *insatisfaisable* si pour toute interprétation I, $[\phi]_I = 0$

Un *théorème* est une formule valide.

Exemple 3.7
$\neg(p \implies p)$ est insatisfaisable :

p	$p \implies p$	$\neg(p \implies p)$
0	1	0
1	1	0

$(\neg p \implies \neg p)$ est une tautologie :

p	$\neg p$	$\neg p \implies \neg p$
0	1	1
1	0	1

et $p \implies q$ est satisfaisable mais non valide :

p	q	$p \implies q$
0	0	1
0	1	1
1	0	0
1	1	1

Remarque : ϕ est valide si et seulement si $\neg\phi$ est insatisfaisable.

3.4 Système de déduction, preuve

Un système de preuve permet de montrer que des formules sont valides en ne manipulant que la *syntaxe*. Les règles de déductions du système de preuve sont définies relativement à la *sémantique* de la logique. C'est là le point clé de la logique : elle permet de raisonner en faisant abstraction de la sémantique, c'est à dire en raisonnant indépendament d'une interprétation particulière.

3.4.1 Représentation du raisonnement : les séquents

Définition 3.8 Un *séquent* est un couple (Γ, ϕ) où Γ est un ensemble fini de formules et ϕ est une formule. ♦

Un séquent formalise le concept de raisonnement : si Γ est vrai (hypothèse) alors ϕ est vraie (conséquence).

Définition 3.9 Un séquent (Γ, ϕ) est vrai dans une interprétation I si

$$(\text{pour toute } \psi \in \Gamma,\ [\psi]_I = 1) \implies [\phi]_I = 1$$

ou encore :

$$(\psi \in \Gamma \implies [\psi]_I = 1) \implies [\phi]_I = 1$$

♦

Un séquent est *valide* s'il est vrai pour toute interprétation I. On note : $\Gamma \models \phi$ lorsque le séquent (Γ, ϕ) est valide. Ceci indique que ϕ est une *conséquence sémantique* de Γ.

Exemple 3.10
Montrer que ψ est vraie dans I (resp. valide) si le séquent $(\emptyset, \psi)$ est vrai dans I (resp. valide).

Les séquents suivants sont-ils valides ?
— $(\emptyset,\ p \implies q)$
— $(\{p, p \implies q\},\ q)$

♦

Proposition 3.11 $\{\psi_1, \ldots, \psi_n\} \models \phi$ ssi $\emptyset \models \psi_n \implies (\psi_{n-1} \implies (\cdots(\psi_1 \implies \phi)\cdots))$ ♦

Preuve. On montre : « $\Gamma, \psi \models \phi$ si et seulement si $\Gamma \models (\psi \implies \phi)$ ». Une induction sur n suffit à conclure la preuve.

Soit I une interprétation. On a :

$$[\psi]_I = 1 \implies [\phi]_I = 1 \quad \text{ssi} \quad [\psi \implies \phi]_I = 1$$

Alors :

$$(\psi' \in \Gamma \implies [\psi']_I = 1) \implies [\psi]_I = 1 \implies [\phi]_I = 1$$
ssi
$$(\psi' \in \Gamma \implies [\psi']_I = 1) \implies [\psi \implies \phi]_I = 1$$

♦

3.4.2 Démonstrations logiques

Une démonstration logique consiste à déduire des séquents dits *prouvables* ou encore *dérivables* en utilisant des règles de déductions. Ces règles de déduction garantissent (comme nous le démontrons plus loin) que les séquents prouvables sont exactement les séquents valides.

Définition 3.12 Un séquent est *prouvable*, noté $\Gamma \vdash \phi$ s'il est construit en utilisant les règles suivantes un nombre fini de fois.

1. utilisation d'une hypothèse : $\dfrac{}{\Gamma, \psi \vdash \psi}\ (hypothese)$
2. augmentation des hypothèses : $\dfrac{\Gamma \vdash \phi \quad \psi \notin \Gamma}{\Gamma, \psi \vdash \phi}\ (augmentation)$
3. détachement : $\dfrac{\Gamma \vdash (\phi \implies \phi') \quad \Gamma \vdash \phi}{\Gamma \vdash \phi'}\ (modus\ ponens)$
4. retrait d'une hypothèse : $\dfrac{\Gamma, \psi \vdash \phi}{\Gamma \vdash (\psi \implies \phi)}\ (synthese)$
5. double négation : $\dfrac{\Gamma \vdash \phi}{\Gamma \vdash \neg\neg\phi}$ et $\dfrac{\Gamma \vdash \neg\neg\phi}{\Gamma \vdash \phi}\ (double\ negation)$
6. raisonnement par l'absurde : $\dfrac{\Gamma, \psi \vdash \phi \quad \text{et} \quad \Gamma, \psi \vdash \neg\phi}{\Gamma \vdash \neg\psi}\ (contradiction)$

♦

Une *démonstration* (ou *preuve*) d'un séquent prouvable $\Gamma \vdash \phi$ est une suite finie de séquents prouvables $\Gamma_i \vdash \phi_i$ telle que :
— le dernier est $\Gamma \vdash \phi$
— tout séquent de la suite d'obtient en appliquant l'une de règles aux séquents qui le précèdent dans la suite.

Exemple 3.13
Démonstration : $\emptyset \vdash (p \implies p)$

$$\dfrac{\dfrac{}{p \vdash p}\ (hypothese)}{\emptyset \vdash (p \implies p)}\ (synthese)$$

Démonstration : $\emptyset \vdash (p \implies (q \implies p))$

$$\dfrac{\dfrac{\dfrac{}{p, q \vdash p}\ (hypothese)}{p \vdash (q \implies p)}\ (synthese)}{\emptyset \vdash (p \implies (q \implies p))}\ (synthese)$$

Démonstration : $p \vdash (\neg p \implies q)$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p, \neg p, \neg q \vdash p}\,(hypothese) \qquad \dfrac{}{p, \neg p, \neg q \vdash \neg p}\,(hypothese)}{p, \neg p \vdash \neg\neg q}\,(contradiction)}{p, \neg p \vdash q}\,(double\ negation)}{p \vdash (\neg p \implies q)}\,(synthese)$$

♦

Théorème 3.14 (Complétude et cohérence) $\Gamma \models \phi$ si et seulement si $\Gamma \vdash \phi$ ♦

Preuve. On ne donne ici que la preuve de cohérence : ≪ tout séquent prouvable est valide : si $\Gamma \vdash \phi$ alors $\Gamma \models \phi$ ≫.

On montre par induction sur la longueur des preuves que si $\Gamma \vdash \phi$ est prouvable alors $\Gamma \models \phi$ est valide. L'induction est réalisée sur les règles du calcul des séquents.

Supposons donc que $\Gamma \vdash \phi$ est obtenu après application d'une règle :

(hypothèse) alors, $\phi \in \Gamma$. Par conséquent, si pour tout $\psi \in \Gamma$ $[\psi]_I = 1$, alors $[\phi]_I = 1$. Alors, $\Gamma \models \phi$.

(augmentation) alors, soit $\Gamma' = \Gamma - \{\psi\}$, $\Gamma' \vdash \phi$. Par induction, $\Gamma' \models \phi$, donc :

$$(\zeta \in \Gamma' \implies [\zeta]_I = 1) \implies [\phi]_I = 1$$

Or :

$$(\zeta \in \Gamma' \cup \{\psi\} \implies [\zeta]_I = 1) \implies (\zeta \in \Gamma' \implies [\zeta]_I = 1)$$

d'où $\Gamma \models \phi$.

(modus ponens) Si $\Gamma \vdash \phi'$, alors $\Gamma \vdash (\phi \implies \phi')$ et $\Gamma \vdash \phi$. Par hypothèse d'induction, $\Gamma \models (\phi \implies \phi')$ et $\Gamma \models \phi$. Alors, $(\zeta \in \Gamma \implies [\zeta]_I = 1)$ implique à la fois $[\phi \implies \phi']_I = 1$ et $[\phi]_I = 1$. Alors, $[\phi']_I = 1$, d'où $\Gamma \models \phi'$.

(synthèse) Si $\Gamma \vdash (\psi \implies \phi)$, alors $\Gamma, \psi \vdash \phi$. Par hypothèse d'induction, $\Gamma, \psi \models \phi$. Si pour toute $\zeta \in \Gamma$, $[\zeta]_I = 1$, il vient :
— si $[\psi]_I = 1$, alors $[\phi]_I = 1$ donc $[\psi \implies \phi]_I = 1$
— si $[\psi]_I = 0$, alors $[\psi \implies \phi]_I = 1$
Donc : $\Gamma \models (\psi \implies \phi)$.

(double negation) Si $\Gamma \vdash \phi$ est obtenu par double négation, sachant que $[\phi]_I = [\neg\neg\phi]_I$, il vient $\Gamma \models \phi$ ssi $\Gamma \models \neg\neg\phi$.

(contradiction) Si $\Gamma \vdash \neg\psi$ alors $\Gamma, \psi \vdash \phi$ et $\Gamma, \psi \vdash \neg\phi$. Par hypothèse d'induction : $\Gamma, \psi \models \phi$ et $\Gamma, \psi \models \neg\phi$. Si pour toute $\zeta \in \Gamma$, $[\zeta]_I = 1$, on ne peut pas avoir $[\psi]_I = 1$, sous peine d'avoir $[\phi]_I = [\neg\phi]_I = 1$. Donc, $[\psi]_I = 0$ et il vient $\Gamma \models \neg\psi$.

♦

Le théorème 3.14 énonce un résultat fondamental. Il signifie que la logique permet de réaliser des preuves de manières purement syntaxique, en appliquant des règles de déduction, sans se soucier aucunement de la sémantique des formules manipulées. En particulier, ces preuves peuvent être calculées par des algorithmes.

3.5 Connecteurs logiques supplémentaires

On utilisera en pratique également les symboles $\wedge$ (« et »), $\vee$ (»ou ») et $\Leftrightarrow$ (« équivaut à »). La définition 3.2 est donc augmentée de la règle :

5. si ϕ et ψ sont des formules, alors $\phi \wedge \psi$, $\phi \vee \psi$ et $\phi \Leftrightarrow \psi$ sont des formules.

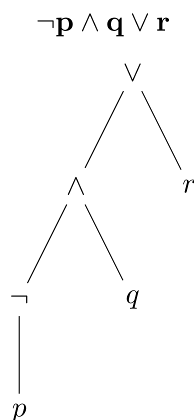
Les connecteurs logiques ont la priorité suivante (du plus prioritaire au moins prioritaire) :

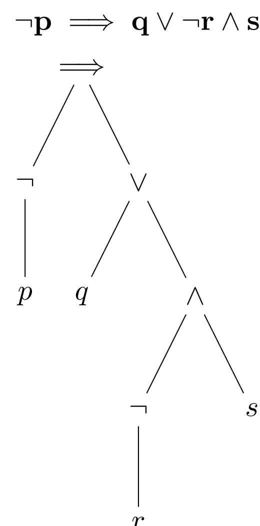
$$\neg \quad \wedge \quad \vee \quad \begin{matrix} \Longrightarrow \\ \Longleftrightarrow \end{matrix}$$

Exemple 3.15
La formule $\neg p \wedge q \vee r$ se lit $((\neg p) \wedge q) \vee r$, et la formule $\neg p \implies q \vee \neg r \wedge s$ se lit $(\neg p) \implies (q \vee ((\neg r) \wedge s))$. ♦

La *structure* d'une formule correspond à son arbre syntaxique, obtenu en tenant compte de la priorité des opérateurs.

Exemple 3.16
Structures des formules de l'exemple précédent :





♦

Ces connecteurs ont les mêmes propriétés d'associativité, de commutativité et de distributivité que les opérateurs correspondants de l'algèbre de Boole :

1. Distributivité de $\wedge$ par rapport à $\vee$: $\phi \wedge (\psi \vee \psi')$ équivaut à $\phi \wedge \psi \;\vee\; \phi \wedge \psi'$
2. Distributivité de $\vee$ par rapport à $\wedge$: $\phi \vee (\psi \wedge \psi')$ équivaut à $(\phi \vee \psi) \wedge (\phi \vee \psi')$
3. Associativité de $\wedge$: $\phi \wedge (\psi \wedge \psi')$ équivaut à $(\phi \wedge \psi) \wedge \psi'$
4. Associativité de $\vee$: $\phi \vee (\psi \vee \psi')$ équivaut à $(\phi \vee \psi) \vee \psi'$
5. Commutativité de $\wedge$: $\phi \wedge \psi$ équivaut à $\psi \wedge \phi$

6. Commutativité de $\vee$: $\phi \vee \psi$ équivaut à $\psi \vee \phi$

La valeur de vérité $[\phi]_I$ est étendue pour ces opérateurs :
— si $\phi = (\psi \wedge \psi')$ alors $[\phi]_I = [\psi]_I \cdot [\psi']_I$
— si $\phi = (\psi \vee \psi')$ alors $[\phi]_I = [\psi]_I + [\psi']_I$
— si $\phi = (\psi \iff \psi')$ alors $[\phi]_I = ([\psi]_I = [\psi']_I)$
Leur sémantique peut également être donnée par les tables de vérité :

$\wedge$	0	1
0	0	0
1	0	1

$\vee$	0	1
0	0	1
1	1	1

$\iff$	0	1
0	1	0
1	0	1

Enfin, quelques équivalences utiles à connaitre :
— $\neg\neg\phi$ équivaut à ϕ
— $\phi \implies \psi$ équivaut à $\neg\phi \vee \psi$
— $\phi \Leftrightarrow \psi$ équivaut à $(\phi \implies \psi) \wedge (\psi \implies \phi)$
— $\phi \vee \psi$ équivaut à $\neg(\neg\phi \wedge \neg\psi)$
— $\phi \wedge \psi$ équivaut à $\neg(\neg\phi \vee \neg\psi)$
Les deux dernière affirmations sont les lois de « de Morgan ».

3.6 Formes normales

Soit ϕ une formule de la logique propositionnelle sur un ensemble $P = \{p_1, \ldots, p_n\}$ de variables propositionnelles. La formule ϕ caractérise la *fonction booléenne* $F_\phi : \mathbb{B}^n \to \mathbb{N}$ qui associe à une interprétation I la valeur de vérité $[\phi]_I$ de ϕ dans I.

Exemple 3.17
Par exemple, $p_1 \wedge p_2$ caractérique la fonction F qui associe 1 à la valuation $(1, 1)$ et 0 aux valuations $(0, 0)$, $(0, 1)$ et $(1, 0)$. ♦

Une formule ϕ est valide, si F_ϕ est la fonction constante 1. Inversement, ϕ est insatisfaisable si F_ϕ est la fonction constante 0. Une formule ϕ est satisfaisable, s'il existe I telle que $F_\phi(I) = 1$. Deux formules ϕ et ψ sont *équivalentes* si $[\phi]_I = [\psi]_I$ pour toute interprétation I, c'est à dire si F_ϕ et F_ψ sont la même fonction.

Une fonction booléenne donnée peut être caractérisée par plusieurs formules. Par exemple, la fonction F qui associe 1 à la valuation $(1, 1)$ et 0 aux autres valuations est caractérisée par la formule $p_1 \wedge p_2$, mais également par la formule $\neg(\neg p_1 \vee \neg p_2)$, et aussi par la formule $p_1 \wedge (p_1 \iff p_2)$. On s'intéresse aux *formes canoniques* pour les formules de la logique propositionnelle.

3.6.1 Forme normale négative

Un *litéral* est une variable propositionnelle (ex : p), ou la négation d'une variable propositionnelle (ex : $\neg p$).

Définition 3.18 Une formule ϕ est en *forme normale négative* si et seulement si elle n'utilise que les connecteurs $\wedge$ et $\vee$, et des litéraux. ♦

Exemple 3.19
Les formules $p_1 \wedge (\neg p_2 \vee p_3)$ et $\neg p_1 \vee p_2 \wedge \neg p_3$ sont en forme normale négative. Mais $\neg(p_1 \wedge p_2)$ ne l'est pas : la négation ne porte pas sur une variable propositionnelle. ♦

Proposition 3.20 Toute formule de la logique propositionnelle admet une formule équivalente en forme normale négative. ♦

En utilisant les équivalences de la section 3.5, il est possible d'éliminer tous les connecteurs $\implies$ et $\iff$. Il reste alors à ≪descendre≫ les négations sur les variables propositionnelles en utilisant les lois de De Morgan.

Exemple 3.21
On commencer par éliminer l'implication de la formule $p \implies \neg(\neg q \vee r)$ ce qui donne $\neg p \vee \neg(\neg q \vee r)$. Puis on descend la négation en utilisant les lois de De Morgan pour obtenir : $\neg p \vee \neg\neg q \wedge \neg r$. Il ne reste qu'à éliminer la double négation : $\neg p \vee q \wedge \neg r$. ♦

3.6.2 Forme normale conjonctive

Une *clause* est une formule de la logique propositionnelle qui n'utilise que le connecteur $\vee$ et des litéraux.

Exemple 3.22
Les formules p, $\neg q$ et $p \vee q \vee \neg r$ sont des clauses. Mais les formules $p \wedge q$, $\neg(p \vee \neg q)$ et $p \vee (q \wedge \neg r)$ n'en sont pas. ♦

On peut toujours supposer qu'une variable apparaît une seule fois dans une clause puisque $p \vee p \equiv p$ et $p \vee \neg p \equiv true$.

Définition 3.23 Une formule de la logique propositionnelle est en *forme normale conjonctive* si elle est une conjonction de clauses. ♦

Exemple 3.24
Par exemple, $(p_1 \vee p_2) \wedge \neg p_3 \wedge (\neg p_1 \vee p_2 \vee \neg p_4)$ est en forme normale conjonctive. ♦

Proposition 3.25 Toute formule de la logique propositionnelle admet une formule équivalente en forme normale conjonctive. ♦

Il existe plusieurs algorithmes pour calculer une formule en forme normale conjonctive équivalente à une formule ϕ de la logique propositionnelle. Une première solution consiste à mettre ϕ sous forme normale négative, puis à utiliser les lois de distributivité (voir section 3.5).

Exemple 3.26
La forme normale négative de $p \implies \neg(\neg q \vee r)$ est $\neg p \vee (q \wedge \neg r)$ (voir exemple 3.21). Cette formule n'est pas en forme normale conjonctive, mais en distribuant $\vee$ sur $\wedge$ on obtient une formule en forme normale conjonctive : $(\neg p \vee q) \wedge (\neg p \vee \neg r)$. ♦

Une autre approche consiste à utiliser la table de vérité de ϕ. On s'intéresse aux valuations pour lesquelles ϕ est fausse. La négation d'une valuation s'exprimer sous la forme d'une clause. La conjonction de ces clauses donne une formule en forme normale conjonctive qui est équivalente à ϕ.

Exemple 3.27
Table de vérité de $\neg p \vee (q \wedge \neg r)$:

p	q	r	$\neg p \vee (q \wedge \neg r)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

On cherche une formule en forme normale conjonctive qui exprime que la formule $\neg p \vee (q \wedge \neg r)$ est vraie, sauf pour les valuations $(1, 0, 0)$, $(1, 0, 1)$ et $(1, 1, 1)$. La clause $\neg p \vee q \vee r$ est vraie sauf pour la valuation $(1, 0, 0)$. De même la clause $\neg p \vee q \vee \neg r$ est vraie sauf pour la valuation $(1, 0, 1)$. Enfin, la clause $\neg p \vee \neg q \vee \neg r$ est vraie sauf pour la valuation $(1, 1, 1)$.

En prenant la conjonction de ces trois clauses : $(\neg p \vee q \vee r) \wedge (\neg p \vee q \vee \neg r) \wedge (\neg p \vee \neg q \vee \neg r)$, on obtient une formule en forme normale conjonctive, qui est équivalente à la formuel de départ $\neg p \vee (\neg q \wedge \neg r)$. ♦

Ces deux approches ont néanmoins le défaut de produire des formules exponentiellement plus grosses que la formule de départ. La transformation de Tseitin permet d'obtenir une forme normale conjonctive de taille linéairement plus grosse que la formule de départ.

3.6.3 Forme normale disjonctive

Une *conjonction élémentaire* est une formule constituée d'une conjonction de litéraux.

Définition 3.28 Une formule est en *forme normale disjonctive* si elle consiste en la disjonction de conjonctions élémentaires. ♦

Exemple 3.29
Les formules $p \vee \neg q$ et $(p \wedge \neg q) \vee (\neg q \wedge r)$ sont en forme normale disjonctive. ♦

Proposition 3.30 Toute formule de la logique propositionnelle admet une formule équivalente en forme normale disjonctive. ♦

On peut aisément construire une formule en forme normale disjonctive, équivalente à une formule ϕ donnée. Pour cela, il suffit de prendre la table de vérité de ϕ. Chaque valuation qui satisfait ϕ correspond à une conjonction élémentaire. La forme normale disjonctive est la disjonction de ces conjonctions élémentaires.

Exemple 3.31
Table de vérité de $\neg p \vee (q \wedge \neg r)$:

p	q	r	$\neg p \vee (q \wedge \neg r)$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Les valuations qui rendent la formule vraie correspondent aux cinq conjonctions élémentaires suivantes : $\neg p \wedge \neg q \wedge \neg r$, $\neg p \wedge \neg q \wedge r$, $\neg p \wedge q \wedge \neg r$, $\neg p \wedge q \wedge r$ et $p \wedge q \wedge \neg r$.

Forme normale disjonctive de $\neg p \vee (\neg q \wedge \neg r)$ s'obtient en prenant la disjonction de ces conjonctions élémentaire :

$$
\begin{aligned}
&(\neg p \wedge \neg q \wedge \neg r) \\
\vee&(\neg p \wedge \neg q \wedge r) \\
\vee&(\neg p \wedge q \wedge \neg r) \\
\vee&(\neg p \wedge q \wedge r) \\
\vee&(p \wedge q \wedge \neg r)
\end{aligned}
$$

♦

Cette approche produit une forme normale disjonctive qui est exponentiellement plus grosse que la formule de départ. Mais c'est inévitable : il existe des formules dont toutes les formes disjonctives équivalentes sont exponentiellement plus grosses.

Chapitre 4

Logique des prédicats

La logique propositionnelle ne permet pas de décrire tous les raisonnements mathématiques. Par exemple, on ne peut pas exprimer une propriété comme :

« tout nombre premier strictement supérieur à 2 est impair »

Cette propriété est expressible en logique des prédicats :

$$\forall x.\ ((premier(x) \wedge x > 2) \implies impair(x))$$

La logique des prédicats introduit :

- des fonctions et des prédicats sur les objets : $>$, $premier$ et $impair$ qui permettent d'exprimer des propriétés de ces objets.
- les quantificateurs : $\forall$ et $\exists$. Ces quantificateurs permettent d'exprimer des propriétés d'ensembles (infinis) d'objets.

La logique introduit une distinction forte entre *syntaxe* et *sémantique*. La syntaxe décrit quelles sont les suites de symboles qui constituent des formules. La sémantique donne une valeur aux formules. Par exemple, $premier(x)$ dans la formule ci-dessus est l'application d'un prédicat, nommé $premier$, à une variable, nommée x. Notre sens commun lui attribue implicitement la signification « x est un nombre premier ». Mais dans un contexte différent, où par exemple x serait une séquence d'entiers $[4; 3; 2; 1; 0]$, alors ce même sens commun donnerait à $premier(x)$ la sémantique de « premier élément de la séquence x », c'est à dire 4 dans cet exemple. Une même formule, ici $premier(x)$ peut donc avoir des sémantiques différentes dans des contextes différents.

4.1 Syntaxe

4.1.1 Langage

Définition 4.1 Soit $\mathcal{F}$ un ensemble de *symboles de fonctions*. À chaque symbole $f \in \mathcal{F}$ on associe son arité $a(f) \in \mathbb{N}$. Si $a(f) = 0$, alors f est un *symbole de constante*. On note C l'ensemble des constantes.

Soit $\mathcal{R}$ un ensemble de *symboles de relations*. À chaque symbole $R \in \mathcal{R}$ est associée sont arité $a(R)$. Si $a(R) = 0$, alors R est une *variable propositionnelle*.

Un *langage* $\mathcal{L} = (\mathcal{F}, \mathcal{R})$. ♦

Un langage est donc un vocabulaire permettant d'exprimer des relations sur des objets. Une formule sur un langage donné ne pourra utiliser que les éléments de ce vocabulaire.

Définition 4.2 Soit X un ensemble de *symboles de variables* et $\mathcal{L}$ un langage.
L'ensemble des *termes* est inductivement défini par :

(B) les symboles de constantes C et de variables X sont des termes

(I) si $f \in \mathcal{F}$ et $t_1, \ldots, t_{a(f)}$ sont des termes, alors $f(t_1, \ldots, t_{a(f)})$ est un terme.

♦

Exemple 4.3
Le langage $\mathcal{L}_{arith} = (\{0, 1, +, *\}, \{\leq\})$ est celui de l'arithmétique entière standard. Par exemple, $1+1+1$ est un terme (qui se décompose en $+(1, +(1, 1))$). De même si $x \in X$, alors $x * x * x + x + x + 1$ est un terme ($x^3 + 2x + 1$). Par contre, $1 - 0$ n'est pas un terme car $-$ n'est pas un symbole de fonction dans ce langage.

Le langage $\mathcal{L}_{Presburger} = (\{0, 1, +\}, \{\leq\})$ est celui de l'arithmétique de Presburger. Dans ce langage, $1 + 1 + 1$ est un terme, mais $x * x * x + x + x + 1$ n'en est pas un. Par contre, $x + x + y$ en est un ($2x + y$). ♦

Exemple 4.4
Un TAD définit un langage. Par exemple :

```
sort queue, elmt

empty : queue
push : queue * elmt -> queue
pop : queue -> queue
top : queue -> elmt
is_empty : queue -> bool
```

définit $\mathcal{L}_{queue} = (\{\mathtt{empty}, \mathtt{push}, \mathtt{pop}, \mathtt{top}\}, \{\mathtt{is_empty}\})$. Notons que `empty` est une constante.

Si $e, p \in X$, alors $\mathtt{push}(\mathtt{empty}, e)$ est un terme. De même $\mathtt{pop}(\mathtt{push}(\mathtt{empty}, e))$ et $\mathtt{top}(p)$.

Remarque : la notion de type n'est pas prise en compte ici, donc $\mathtt{pop}(\mathtt{top}(\mathtt{empty}))$ est un terme. ♦

4.1.2 Formules

Définition 4.5 Soit $\mathcal{L} = (\mathcal{F}, \mathcal{R})$ un langage et X un ensemble de symboles de variables. L'ensemble des formules de la *logique des prédicats* est inductivement défini par :

- **(B)** Si $R \in \mathcal{R}$ est un symbole de relation et $t_1, \ldots, t_{a(R)}$ sont des termes sur $\mathcal{L}$, alors $R(t_1, \ldots, t_{a(R)})$ est une formule (dite *atomique*)
- **(I1)** Si ϕ et ψ sont des formules, alors $\neg\phi$, $\phi \implies \psi$, $\phi \Leftrightarrow \psi$, $\phi \wedge \psi$ et $\phi \vee \psi$ sont des formules
- **(I2)** Si ϕ est une formule et $x \in X$, alors $\exists x.\ \phi$ et $\forall x.\ \phi$ sont des formules.

♦

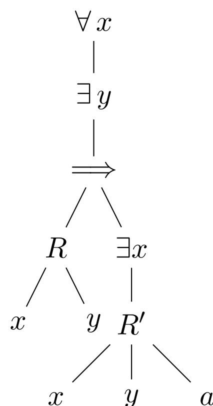
Exemple 4.6
$1 + x \leq 0$ est une formule sur $\mathcal{L}_{Presburger}$. $\texttt{is_empty}(\texttt{empty})$ est une formule sur $\mathcal{L}_{queue}$
Et $\neg(x \leq 1) \implies \texttt{top}(\texttt{push}(\texttt{push}(p, e1), e2)) \leq 1 + x$ est une formule sur l'union de ces deux langages. ♦

On applique les règles de priorité suivantes sur les connecteurs logiques (du plus prioritaire au moins prioritaire) :

$$\neg \qquad \wedge \qquad \vee \qquad \begin{array}{c}\implies \\ \iff\end{array} \qquad \begin{array}{c}\exists \\ \forall\end{array}$$

De même que pour les formules de la logique propositionnelle, la structure d'une formule de la logique des prédicats est l'arbre syntaxique de la formule, tenant compte de la priorité des opérateurs :

Exemple 4.7
$\forall x.\ \exists y.\ R(x, y) \implies \exists x R'(x, y, a)$ a la structure suivante :



♦

Une *sous-formule* de ϕ est n'importe quel sous-arbre de ϕ qui représente une formule.

Exemple 4.8
Quelles sont les sous-formules de $\forall x \exists y\, R(x, y) \implies \exists x R'(x, y, a)$ de l'exemple précédent ? ♦

Une occurrence d'une variable dans une formule est *libre* si elle est sous la portée d'aucun quantificateur $\exists$ ou $\forall$. C'est à dire qu'une *occurrence libre* d'une variable x dans ϕ est une feuille de l'arbre de ϕ telle qu'il n'y a pas de nœud $\exists x$ ou $\forall x$ sur la branche de x à la racine.

Sinon, il s'agit d'une *occurrence liée* de x dans ϕ.

Exemple 4.9
Quelles sont les occurrences libres et liées dans $\forall x \exists y\, R(x, y) \implies \exists x R'(x, y, a)$? ♦

Une variable est *libre* dans ϕ si elle a au moins une occurrence libre. Elle est *liée* si toutes ses occurrences sont liées.

4.2 Sémantique

4.2.1 Structure : interprétation d'un langage

Un langage $\mathcal{L}$ a de multiples interprétations possibles. Une $\mathcal{L}$-structure donne une sémantique aux symboles de fonctions et de relations.

Définition 4.10 Une $\mathcal{L}$-structure est un triplet $S = (\mathbb{D}, \rho, \gamma)$ où :
— $\mathbb{D}$ est un ensemble non vide (le domaine)
— ρ associe à chaque $R \in \mathcal{R}$ un sous-ensemble R_S de $\mathbb{D}^{a(R)}$
— γ associe à chaque $f \in \mathcal{F}$ une fonction $f_S : \mathbb{D}^{a(f)} \to \mathbb{D}$. En particulier, à chaque constante a est associée une valeur a_S.

♦

Exemple 4.11
On définit la sémantique du TAD `queue` par la $\mathcal{L}$-structure S_p suivante. Le domaine est celui des mots finis sur l'alphabet Σ des symboles de pile, étendu par l'élément « indéfini » : $\mathbb{D} = \Sigma^* \cup \{\bot\}$.

Sémantique des constructeurs :
— `empty` est le mot vide ε
— `push`(q, x) a la sémantique de la concaténation : $q \cdot x$

Sémantique de `top` :

$$f_{\texttt{top}}(q) = \begin{cases} \bot & \text{si } q = \varepsilon \\ x & \text{si, } q = x \cdot q',\ x \in \Sigma,\ q' \in \Sigma^* \end{cases}$$

Les axiomes servent à définir la sémantique des fonctions et relations :

```
top(empty)          indéfini
top(push(empty,x))  = x
top(push(q,x))      = top(q)
```

On peut montrer l'équivalence des deux définitions. ♦

4.2.2 Valuation

Une *valuation* est une application de l'ensemble des symboles de variables X dans $\mathbb{D}$. Une valuation donne donc une valeur à chacune des variables. Notons qu'il n'y a pas ici de notion de type de variable.

Deux valuations v, v' sont *congruentes* sur $Y \subseteq X$, noté $v =_Y v'$ si elles coïncident sur les variables de Y : $v(y) = v'(y)$ pour tout $y \in Y$.

Définition 4.12 La *valeur* $[t]_{S,v} \in \mathbb{D}$ d'un terme t dans une $\mathcal{L}$-structure S et valuation v est définie par :

— si $t = a \in C$, alors $[t]_{S,v} = a_S$
— si $t = x \in X$, alors $[t]_{S,v} = v(x)$
— si $t = f(t_1, \ldots, t_n)$ alors $[t]_{S,v} = f_S([t_1]_{S,v}, \ldots, [t_n]_{S,v})$.

♦

Exemple 4.13
Soit q une variable de pile et $v(q) = abcde$ une valuation de q. On associe au symbole de fonction `top` la fonction $f_{\texttt{top}} : \Sigma^* \to (\Sigma \cup \{\perp\})$ introduite dans l'exemple précédent.

Alors $[\texttt{top}(q)]_{S_p,v} = f_{\texttt{top}}([q]_{S_p,v}) = f_{\texttt{top}}(v(q)) = f_{\texttt{top}}(abcde) = a$. ♦

4.2.3 Sémantique des formules

Définition 4.14 Soit S une $\mathcal{L}$-structure et v une valuation. La *valeur de vérité* d'une formule ϕ est donnée par $[\phi]_{S,v} \in \mathbb{B}$:

— si $\phi = R(t_1, \ldots, t_n)$ alors $[\phi]_{S,v} = 1$ si et seulement si $([t_1]_{S,v}, \ldots, [t_n]_{S,v}) \in R_S$
Remarque : si $a(R) = 0$, alors $[\phi]_{S,v} = 1$ si $R_S \neq \emptyset$ et $[\phi]_{S,v} = 0$ sinon.
— $[\neg\phi]_{S,v} = \overline{[\phi]_{S,v}}$
— $[\phi \implies \psi]_{S,v} = 1$ ssi $[\phi]_{S,v} \leq [\psi]_{S,v}$
— $[\phi \wedge \psi]_{S,v} = 1$ ssi $[\phi]_{S,v} = 1$ et $[\psi]_{S,v} = 1$
— $[\phi \vee \psi]_{S,v} = 1$ ssi $[\phi]_{S,v} = 1$ ou $[\psi]_{S,v} = 1$
— $[\exists x\, \phi]_{S,v} = 1$ ssi il existe une valuation w telle que $w =_{X-\{x\}} v$ et $[\phi]_{S,w} = 1$
— $[\forall x\, \phi]_{S,v} = 1$ ssi pour toute valuation w tq $w =_{X-\{x\}} v$ nous avons $[\phi]_{S,w} = 1$

♦

Exemple 4.15
Soit ϕ la formule $\exists x\, x * y = a$, et soit v la valuation $v(x) = 1$ et $v(y) = 2$. La $\mathcal{L}$-structure S donne $a_S = 5$. Les autres symboles : $*$ et $=$ sont interprétés de façon usuelle.

$$
\begin{aligned}
[\exists x\, x * y = a]_{S,v} = 1 \quad & \text{ssi} \quad \text{il existe } w \text{ tq } w(y) = 2 \text{ et } [x * y = a]_{S,w} = 1 \\
& \text{ssi} \quad \text{il existe } w \text{ tq } w(y) = 2 \text{ et } ([x * y]_{S,w}, [a]_{S,w}) \in =_S \\
& \text{ssi} \quad \text{il existe } w \text{ tq } w(y) = 2 \text{ et } (*_S([x]_{S,w}, [y]_{S,w}), a_S) \in =_S \\
& \text{ssi} \quad \text{il existe } w \text{ tq } w(y) = 2 \text{ et } (*_S(w(x), w(y)), 5) \in =_S \\
& \text{ssi} \quad \text{il existe } w \text{ tq } (*_S(w(x), 2), 5) \in =_S \\
& \text{ssi} \quad \text{il existe } w \text{ tq } w(x) * 2 = 5
\end{aligned}
$$

Dans l'arithmétique entière $\mathbb{D} = \mathbb{N}$, il n'existe pas de telle valuation w. Dans l'arithmétique réelle, il suffit de prendre w telle que $w(x) = 2.5$. ♦

Deux formules ϕ et ψ sont *équivalentes* si pour toute $\mathcal{L}$-structure S et pour toute valuation v, $[\phi]_{S,v} = [\psi]_{S,v}$. Notamment :

$$\forall x\, \phi \;\equiv\; \neg\exists x\, \neg\phi \qquad \text{et} \qquad \exists x\, \phi \;\equiv\; \neg\forall x\, \neg\phi$$

Proposition 4.16 Soit Y l'ensemble des variables libres dans ϕ. Si $v =_Y w$ alors $[\phi]_{S,v} = [\phi]_{S,w}$ pour toute $\mathcal{L}$-structure S. ♦

Définition 4.17 Une formule ϕ est :
— *satisfaisable* s'il existe une $\mathcal{L}$-structure S et une valuation v telles que $[\phi]_{S,v} = 1$
— *valide* si elle $[\phi]_{S,v} = 1$ pour toute $\mathcal{L}$-structure S et tout valuation v.

♦

Exemple 4.18
$((\neg\exists x\, P(x)) \;\Leftrightarrow\; (\forall x\, \neg P(x)))$ est universellement valide.
Si S est la structure associées à $\mathbb{R}$, alors $x \leq x + y$ est satisfaisable dans S, mais elle n'est pas valide dans S. ♦

Exemple 4.19
Soit $S = \langle E, \{R, =\}\rangle$ un ensemble muni d'une relation R et du prédicat d'égalité. Écrire une formule qui est valide dans S ssi R est un ordre (resp. total).

Une relation d'ordre est réflexive, transitive et antisymétrique. La formule doit exprimer ces trois propriétés :

$$
\begin{aligned}
& (\forall x\, xRx) \\
& \wedge\, (\forall x \forall y \forall z\, (xRy \;\wedge\; yRz \implies xRz)) \\
& \wedge\, (\forall x \forall y (xRy \wedge yRx \implies x = y)
\end{aligned}
$$

Pour un ordre total, il faut ajouter :

$$\wedge(\forall x \forall y\, (xRy \vee yRx))$$

♦

Théorème 4.20 La satisfaisabilité des formules de la logique des prédicats est indécidable.

4.3 Substitution

On note $\phi[x \leftarrow t]$ la formule obtenue en *substituant* le terme t à toutes les *occurrences libres* de x. Si x n'a aucune occurrence libre dans ϕ, alors $\phi[x \leftarrow t] = \phi$.

On dit que t est *substituable* à x dans ϕ si t est clos (aucune variable) ou toutes les occurrences de variables dans t sont libres dans $\phi[x \leftarrow t]$.

Exemple 4.21
Soit ϕ la formule $(\forall y\, R(x, y, z)) \lor (\forall z\, R'(z))$.

Un terme t est substituable à x dans ϕ ssi y n'a pas d'occurrence dans t :

— Par exemple : $\phi[x \leftarrow y] = (\forall y\, R(y, y, z)) \lor (\forall z\, R'(z))$ et la première occurrence de y devient liée.

— Par contre : $f(x, z)$ est substituable à x, en effet :

$$\phi[x \leftarrow f(x, z)] = (\forall y\, R(f(x, z), y, z)) \lor (\forall z\, R'(z))$$

La proposition suivante donne une sémantique à la substitution syntaxique :

Proposition 4.22 Soit x une variable et t un terme substituable à x dans ϕ. Soit v une valuation et w définie par :

$$w(y) = \begin{cases} [t]_v & \text{si } y = x \\ v(y) & \text{sinon} \end{cases}$$

Alors, $[\phi]_{S,w} = [(\phi[x \leftarrow t])]_{S,v}$ pour toute $\mathcal{L}$-structure S.

Preuve. Par induction sur ϕ.

4.4 Système de déduction, preuve

Le calcul de Gentzen (calcul des séquents) introduits en section 3.4.2 s'étend à la logique des prédicats de la manière suivante.

Définition 4.23 Un séquent est *prouvable*, noté $\Gamma \vdash \phi$ s'il est construit en utilisant les règles de la définition 3.12 et les règles suivantes un nombre fini de fois.

7. introduction de $\forall$: $\dfrac{\Gamma \vdash \phi}{\Gamma \vdash \forall x.\, \phi}$ $(\forall_I)$ pour x *non libre* dans Γ
8. élimination de $\forall$: $\dfrac{\Gamma \vdash \forall x.\, \phi}{\Gamma \vdash \phi[x \leftarrow t]}$ $(\forall_E)$ où t est un terme quelconque

9. introduction de $\exists$: $\dfrac{\Gamma \vdash \phi[x \leftarrow t]}{\Gamma \vdash \exists x.\, \phi}$ $(\exists_I)$ où t est un terme quelconque
10. élimination de $\exists$: $\dfrac{\Gamma \vdash \exists x.\, \phi \qquad \Gamma, \phi \vdash \psi}{\Gamma \vdash \phi}$ $(\exists_E)$ pour x *non libre* dans Γ et ψ.

♦

La règle $(\forall_I)$ indique que prouver ϕ pour un x arbitraire, revient à prouver $\forall x.\, \phi$. Pour que x soit arbitraire, il faut qu'il soit non libre dans Γ. Sans cette contrainte, on peut par exemple prouver le séquent $x = 2 \vdash \forall x.\, x = 2$ qui n'est pas valide (le séquent vaut $false$ pour la valuation $v(x) = 2$) :

$$\dfrac{\dfrac{}{x = 2 \vdash x = 2}\,(Hyp)}{x = 2 \vdash \forall x.\, x = 2}\,(\forall_I)$$

La règle $(\forall_E)$ indique que prouver $\forall x.\, \phi$ permet de prouver ϕ pour une valeur arbitraire t de x. Le terme t peut être une valeur fixe (ex : 2) aussi bien qu'un terme de valeur quelconque (ex : $3 * y + 7$).

La règle $(\exists_I)$ indique que prouver ϕ pour une certaine valeur t de x permet de prouver $\exists x.\, \phi$. Comme précédemment, t est un terme quelconque.

Enfin, la règle $(\exists_E)$ indique qu'en prouvant $\exists x.\, \phi$, c'est à dire qu'on peut choisir x tel que ϕ est vraie, et qu'en prouvant que lorsque ϕ est vraie, ψ est également vraie, alors on peut prouver ψ. Attention, là encore, x ne doit pas être libre dans Γ et ψ sous peine de prouver des séquents non valides comme dans l'exemple ci-dessus.

On peut maintenant énoncer le théorème de complétude prouvé en 1929 par Kurt Gödel.

Théorème 4.24 (Cohérence et complétude) $\Gamma \models \phi$ si et seulement si $\Gamma \vdash \phi$ ♦

C'est à dire que si un séquent est valide alors il est prouvable. Et inversement, si un séquent est prouvable, alors il est valide. Notons que ela ne contredit pas le théorème 4.20 : la complétude (théorème 4.24) dit qu'il existe une preuve, et l'indécidabilité dit qu'il n'existe pas d'algorithme permettant de calculer cette preuve. En particulier, un algorithme qui énumère toute les preuves :

— terminera avec une preuve de validité si $\Gamma \vdash \phi$,

— mais ne terminera pas si $\Gamma \vdash \phi$ n'est pas un séquent valide.

Deuxième partie

Preuve de programmes

Chapitre 5

Motivation pour la preuve de programmes

Pourquoi prouver des programmes ?

(Bank of New York, 1985) Effondrement du cours des bons du Trésor américain constaté en salle des marchés de la BNY.

Investissement à perte dans les métaux précieux pour sauver le capital.

La ≪ chute ≫ du cours est du à un problème de représentation des nombres dans le système informatique de BNY.

Pertes : 20 milliards de dollars

(AT&T, 1990) Pannes en cascade ds relais téléphoniques de AT&T.

Le problème vient d'un `break` manquant :

```
swicth (...) {
case 1: ...
        // break manquant
case 2:
        reboot();
        break;
...
}
```

Pertes : 7.000.000 appels et une réputation à refaire.

(Intel, 1994) Bug dans l'unité de calcul en virgule flottante.

Pertes : échange des processeurs Pentium commercialisés et perte de réputation (AMD, etc).

Conséquence : introduction des méthodes formelles pour le développement des processeurs

(Arianespace/ESA/CNES, 1996) Le vol 501 de la fusée Ariane V se conclût par la destruction du lanceur.

Problème du à un morceau de code d'Ariane IV qui s'exécute par erreur et qui est victime de débordements numériques (caractéristiques physiques très différentes).

Pertes : financières et confiance perdues au profit de la concurrence.

Conséquence : introduction de la sûreté de fonctionnement et des méthodes formelles dans l'aérospatiale

(AECL/CGR MeV, 1985-1987) Le THERAC-25 (traitement des tumeurs par irradiation) délivre des doses jusqu'à 100 fois supérieurs à la normale.

Problèmes :
— Négligences dans les tests du logiciel
— Conditions de concurrence (race condition)
— Dépassement des capacité sur les variables paramétrant les tests de sécurité

Pertes : plusieurs morts et des blessés graves (handicaps importants)

Le *test du logiciel* consiste à fournir certaines valeurs d'entrée au programme et à vérifier que son comportement et les sorties produites sont conformes aux spécifications.

Le nombre de tests à réaliser croît exponentiellement avec :
— le nombre de paramètres d'entrée et de variables du programmes
— la « complexité » du programme : nombre de boucles, nombre d'instructions conditionnelles, complexité des conditions des boucles et des instructions conditionnelles, etc.

En pratique, l'exhaustivité est rapidement hors de portée.

Si le test est indispensable, il demeure en général insuffisant. Il faut donc des techniques visant l'*exhaustivité* pour atteindre un processus de *développement rigoureux du logiciel*. La preuve de programme est l'une des techniques permettant d'y parvenir.

Les *techniques formelles de développement logiciel* sont utilisées dans l'industrie :
— la preuve de programme (la méthode B) est utilisée pour les trains automatiques (ligne 14 à Paris, Orlyval, métro de Toulouse, etc.)
— l'interprétation abstraite (logiciel Astrée) est utilisée chez Airbus pour l'A380 et l'A340 (et les avions à venir)
— le model-checking à base d'abstraction/raffinement (outil SLAM) est utilisé par Microsoft pour le développement de Windows

Chapitre 6

Spécification et sémantique des programmes

6.1 Spécification

— modèle descriptif ou *spécification* : répond à la question QUOI ?
— modèle opérationnel ou *programme* : répond à la question COMMENT ?
— *preuve* : assure la cohérence du programme et de la spécification

La spécification est l'énoncé d'un *but* ou post-condition à atteindre dans un certain *cadre* ou pré-condition. Elle exprime l'idée d'un contrat : si l'utilisateur du programme s'engage à respecter la pré-condition, alors le programmeur garantit que la post-condition est satisfaite par son programme.

Définition 6.1 Une *spécification* d'un programme est un couple (P, R) de formules du 1er ordre sur un langage $\mathcal{L}$ donné. La formule P est appelée la *pré-condition* et R est appelée la *post-condition*. ♦

Exemple 6.2

— Racine par défaut
  — DONNÉES : n entier naturel
  — RÉSULTAT : r entier naturel, racine entière de n
  — PRECOND : $n \in \mathbb{N}$
  — POSTCOND : $r^2 \leq n < (r+1)^2$

— Tri
  — DONNÉES : t un tableau de n entiers de capacité MAX
  — RÉSULTAT : t est trié en ordre croissant
  — PRECOND : $0 \leq n < MAX$
  — POSTCOND : $\forall i.\forall j.(0 \leq i < j < n \Rightarrow t[i] \not\geq t[j])$

6.2 Programmes et sémantique

6.2.1 While programs

Compromis entre :
— expressivité (instructions, structures de données, pointeurs, arithmétique, etc)
— prouvabilité

Dans ce cours, on se limite à un *jeu d'instruction restreint* (mais Turing-complet) et à une *manipulation fonctionnelle* des données (absence d'effet de bord).
En TD : peuve de programmes C

On considère un ensemble de variables X et un langage $\mathcal{L}$.

Définition 6.3 L'ensemble des *while-programs* est inductivement défini par :
— « `skip` » et « `x:=t` » pour $\mathtt{x} \in X$ et `t` un terme sur $(\mathcal{L}, X)$, sont des while-programs
— si S_1 et S_2 sont des while-programs, et B est une formule du 1er ordre sur $(\mathcal{L}, X)$, alors : « S_1;S_2 » et « **if** B **then** S_1 else S_2 **end** » et « **while** B **do** S_1 **end** » sont des while-programs.

♦

Exemple 6.4
Racine carrée par défaut (i.e. $r = \lfloor\sqrt{n}\rfloor$) :

```
r := 0;
while (n ≥ (r+1)²) do
  r := r+1
end
```

♦

6.2.2 Sémantique, correction et terminaison

Considérons un langage $\mathcal{L}$ et une $\mathcal{L}$-structure $S = (\mathbb{D}, \rho, \gamma)$.

Un *environnement* est une valuation des variables du programme.

Exemple 6.5
Pour la racine carrée par défaut, $\mathbb{D} = \mathbb{N}$. Alors, $v : n \mapsto 10,\ r \mapsto 2$ est un environnement.
♦

Définition 6.6 *L'évaluation d'un while-program S* dans un environement v consiste en la transformation de (S, v) en (S', v') comme suit :
— $(\mathtt{skip}, v) \to (\emptyset, v)$
— $(\mathtt{x} := \mathtt{t}, v) \to (\emptyset, v')$ où $v'(y) = \begin{cases} [t]_v & y = x \\ v(y) & \text{sinon} \end{cases}$

— $(S_1; S_2, v) \to (S'_1; S_2, v')$ si $(S_1, v) \to (S'_1, v')$ et $S'_1 \neq \emptyset$
— $(S_1; S_2, v) \to (S_2, v')$ si $(S_1, v) \to (\emptyset, v')$
— $(\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}, v) \to (S_1, v)$ si $[B]_v = 1$
— $(\texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}, v) \to (S_2, v)$ si $[B]_v = 0$
— $(\texttt{while } B \texttt{ do } S \texttt{ end}, v) \to (S\,;\texttt{ while } B \texttt{ do } S \texttt{ end}, v)$ si $[B]_v = 1$
— $(\texttt{while } B \texttt{ do } S \texttt{ end}, v) \to (\emptyset, v)$ si $[B]_v = 0$

♦

Définition 6.7 Une *exécution* d'un while-program S depuis un environnement v est une séquence finie ou infinie d'évaluations :

$$(S, v) \to (S_1, v_1) \to \cdots \to (S_i, v_i) \to \cdots$$

♦

On voit apparaître ici deux notions fondamentales :
— L'*état* d'un programme (S, v) constitué de la prochaine instruction à exécuter S et d'une valeur pour chacune des variables v. La prochaine instruction à exéctuter est généralement représentée par un registre nommée "program counter (pc)" en machine.
— Une *exécution* d'un programme est une suite d'états obtenus par évaluation de l'instruction à exécuter.

Exemple 6.8
On reprend l'exemple de la racine carrée par défaut, on nomme W le while-program qui correspond à la boucle `while`.

Considérons un environnement initial tel que $n = 0$. On a alors :

$$(r := 0; W, 0, -) \to (\texttt{while } n \geq (r+1)^2 \texttt{ do } r := r + 1 \texttt{ end}, 0, 0) \to (\emptyset, 0, 0)$$

Considérons un environnement initial tel que $n = 5$. L'exécution de P est alors la suivante :

$$(r := 0; W, 5, -) \to (W, 5, 0) \to (r := r + 1; W, 5, 0) \to (W, 5, 1) \to (r := r + 1; W, 5, 1) \\ \to (W, 5, 2) \to (\emptyset, 5, 2)$$

♦

La *sémantique* d'un while-program S est l'ensemble de ses exécutions. C'est à dire, l'ensemble des exécutions de S pour tout environnement v.

Une exécution d'un while-program S depuis un environnement v *termine* si elle est finie, c'est à dire s'il existe $n \in \mathbb{N}$ tel que $S_n = \emptyset$.

$$(S, v) \to (S_1, v_1) \to \cdots \to (S_n = \emptyset, v_n)$$

Soit (P, R) une spécification. Une exécution finie d'un while-program S depuis un environnement v tel que $[P]_v = 1$ est *correcte* si $[R]_{v_n} = 1$. C'est à dire qu'à partir d'un environnement qui satisfait la pré-condition, l'exécution de S se termine dans un environnement qui satisfait la post-condition.

Définition 6.9 Une while-program S *termine* si pour tout environnement v, l'exécution de S depuis v termine. S est *correct* pour une spécification (P, R) si pour tout environnement v l'exécution de S depuis v est correcte pour (P, R). ♦

D'après cette définition on voit que pour qu'un programme soit correct et qu'il termine, il faut qu'au long de toute exécution le programme reste dans l'*ensemble des états valides*. Cette approche, basée sur la notion d'état d'un programme, est à la base des techniques de vérification de programmes. Programmer de manière correcte revient justement à réfléchir en terme d'états et à maintenir le programme dans un état correct.

En général, l'ensemble des exécutions d'un programme est infini puisque l'ensemble des environnements est infini (il suffit d'une variable non bornée). Comment prouver qu'un programme termine ? Comment prouver qu'un programme est correcte ?

Théorème 6.10 La terminaison et la correction des while-programs sont des problèmes indécidables. ♦

Il n'existe donc pas de méthode générale qui prouver que tout programme termine ni que tout programme est correcte.

Chapitre 7

Preuve de programmes sans boucles

7.1 Terminaison

Théorème 7.1 Tout while-program ne contenant pas de boucle `while` termine ♦

Preuve. On remarque que pour toute instruction (différente de `while`), si $(S, v) \to (S', v')$ alors $|S'| < |S|$. ♦

7.2 Correction

7.2.1 Calcul de Hoare

But : prouver qu'un programme S satisfait une spécification (P, R). On note cela par un triplet de Hoare :

$$\{P\}\ S\ \{R\}$$

On note $(S, v) \to^+ (S', w)$ s'il existe une séquence *finie* : $(S, v) \to \cdots \to (S', w)$.

Un triplet est *valide* si pour toute valuation v qui satisfait P (i.e. $[P]_v = 1$), si l'exécution de S depuis v termine : $(S, v) \to^+ (\emptyset, w)$, alors w satisfait R (i.e. $[R]_w = 1$).

Exemple 7.2
Le triplet suivant n'est pas valide :

$$\{x \geq 0\}\ \mathtt{x} := \mathtt{x} + \mathtt{1}\ \{x \geq 2\}$$

En effet, la valuation $v : x \mapsto 0$ satisfait $(x \geq 0)$, or $(\mathtt{x} := \mathtt{x} + \mathtt{1}, 0) \to (\emptyset, 1)$ et $w : x \mapsto 1$ ne satisfait pas $(x \geq 2)$. ♦

Définition 7.3 Un triplet $\{P\}\ S\ \{R\}$ est *prouvable* s'il est construit par l'application des règles ci-dessous un nombre fini de fois.

1. skip : $\overline{\{R\}\mathtt{skip}\{R\}}$

2. Affectation : $\overline{\{R[x \leftarrow t]\}x := t\{R\}}$
   où `t` est substituable dans R.
3. Séquence : $\dfrac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}$
4. Conditionnelle : $\dfrac{\{P \wedge B\}S_1\{R\} \quad \{P \wedge \neg B\}S_2\{R\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\ \{R\}}$
5. Conséquence : $\dfrac{P' \implies P \quad \{P\}S\{R\} \quad R \implies R'}{\{P'\}S\{R'\}}$

♦

La règle de conséquence génère des *obligations de preuve*. Il faut en effet prouver que $P' \implies P$ et $R \implies R'$ sont deux formules valides.

Exemple 7.4
Quelques preuves :

$$\overline{\{0 = 0\}\ \texttt{x := 0}\ \{x = 0\}}$$

$$\dfrac{\overline{\{0 = 0 \wedge 0 = 0\}\ \texttt{x := 0}\ \{x = 0 \wedge 0 = 0\}} \qquad \overline{\{x = 0 \wedge 0 = 0\}\ \texttt{y := 0}\ \{x = 0 \wedge y = 0\}}}{\{0 = 0 \wedge 0 = 0\}\ \texttt{x := 0; y := 0}\ \{x = 0 \wedge y = 0\}}$$

$$\dfrac{\mathit{true} \implies 0 = 0 \qquad \overline{\{0 = 0\}\ \texttt{x := 0}\ \{x = 0\}} \qquad x = 0 \implies x \geq 0}{\{\mathit{true}\}\ \texttt{x := 0}\ \{x \geq 0\}}$$

il reste à prouver la validité de $\mathit{true} \implies 0 = 0$ et $x = 0 \implies x \geq 0$ dans la théorie appropriée. ♦

Afin que le calcul de Hoare soit intéressant, il faut que tout triplet prouvable soit valide, et inversement, que tout triplet valide soit prouvable :

Complétude : si $(S, v) \to^+ (\emptyset, w)$ alors pour tout P tel que $[P]_v = 1$, il existe R tel que $[R]_w = 1$ et $\{P\}\ S\ \{R\}$ est un triplet prouvable.

Correction : si $\{P\}\ S\ \{R\}$ est prouvable, alors pour toute valuation v tq $[P]_v = 1$ on a $(S, v) \to^+ (\emptyset, w)$ avec $[R]_w = 1$.

Théorème 7.5 (Complétude et correction) Le calcul de Hoare est correct et relativement complet. ♦

Relativement complet signifie que la complétude du calcul de Hoare repose sur la complétude de la logique sous-jacente. La régle conséquence nécessite de prouver la validité de $P' \implies P$ et de $R \implies R'$. La capacité à réaliser cette preuve dépend de la complétude de la logique sous-jacente (par exemple FOL).

Exemple 7.6
On illustre la correction et la complétude (relative) pour certaines instructions :

Affectation : on a $(x := t, v) \to (\emptyset, w)$ avec :

$$w(y) = \begin{cases} [t]_v & \text{si } y = x \\ v(y) & \text{sinon} \end{cases}$$

et la règle : $\dfrac{}{\{R[x \leftarrow t]\}\ \texttt{x := t}\ \{R\}}$

Si $[(R[x \leftarrow t])]_v = 1$ alors par la propriété 4.22, $[R]_w = 1$.

Conditionnelle : on a :

$$\begin{aligned} (\texttt{if}\, B\, \texttt{then}\, S_1\, \texttt{else}\, S_2\, \texttt{end}, v) &\to (S_1, v) \quad \text{si } [B]_v = 1 \\ (\texttt{if}\, B\, \texttt{then}\, S_1\, \texttt{else}\, S_2\, \texttt{end}, v) &\to (S_2, v) \quad \text{si } [B]_v = 0 \end{aligned}$$

et la règle : $\dfrac{\{P \wedge B\}S_1\{R\} \quad \{P \wedge \neg B\}S_2\{R\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\ \{R\}}$

Supposons que $[B]_v = 1$, alors $[P \wedge B]_v = 1$. Par hypothèse d'induction, $\{P \wedge B\}\, S_1\, \{R\}$ est valide, donc $(S_1, v) \to^+ (\emptyset, w)$ avec $[R]_w = 1$. Il vient donc : $(\texttt{if}\, B\, \texttt{then}\, S_1\, \texttt{else}\, S_2\, \texttt{end}, v) \to^+ (\emptyset, w)$ et $[R]_w = 1$.

Le cas $[B]_v = 0$ est symétrique.

Chapitre 8

Preuve de programmes avec boucles

8.1 Terminaison : variants

Un programme avec boucle `while` ne termine pas nécessairement.

Exemple 8.1

```
x := 0;
while (x ≥ 0) do
  x := x+1
end
```

a pour exécution depuis tout environnement v (soit W le while-program correspondant à l'instruction `while`) :

$$(\mathtt{x} := \mathtt{0}; W, -) \to (W, 0) \to (\mathtt{x} := \mathtt{x} + \mathtt{1}; W, 0) \to (W, 1) \to (\mathtt{x} := \mathtt{x} + \mathtt{1}; W, 1) \to (W, 2) \to \cdots$$

Comment démontre-t-on qu'une suite de configurations :

$$(S, v) \to (S_1, v_1) \to \cdots \to (S_i, v_i) \to \cdots$$

est finie ? On montre qu'elle *converge*.

8.1.1 Ensembles bien fondés

Principe bien connu sur $\mathbb{N}$:

— tout suite monotone (strictement croissante/décroissante)

— et bornée (en haut/bas)

converge.

On généralise à d'autres types de données manipulés par les programmes informatiques.

Une relation d'ordre $\leq$ sur un ensemble E : réflexive, antisymétrique et transitive.

Définition 8.2 Une relation d'ordre $\leq$ sur un ensemble E est *bien fondée* s'il n'y a pas de suite infinie strictement décroissante d'éléments de E. Un *bon ordre* est un ordre total bien fondé. ♦

Caractérisation importante des ensembles bien fondés :

Théorème 8.3 Un ensemble ordonné $(E, \leq)$ est bien fondé si et seulement si toute partie non vide de E admet un élément minimal. ♦

Preuve. Nous montrons la contraposée : $(E, \leq)$ admet une suite infinie strictement décroissante ssi il existe une partie non vide de E n'ayant pas d'élément minimal.

Supposons qu'il existe une suite infinie strictement décroissante $(x_n)_{n\geq 0}$ dans E. Alors, $X = \{x_n \mid n \geq 0\}$ est une partie non-vide de E qui n'a pas d'élément minimal.

Réciproquement, supposons qu'il existe une partie non-vide X de E n'ayant pas d'élément minimal. Puisque X n'a pas d'élément minimal, tout élément x de X est strictement plus grand qu'un autre élément y de X. Il existe donc une fonction $f : X \to X$ vérifiant $f(x) < x$. Il suffit en effet de poser $f(x) = y$ où y est l'un des éléments tels que $y < x$. Soit $x_0 \in X$ (X est non-vide par hypothèse). Pour tout entier $n \geq 0$, on définit $x_n = f^n(x_0)$. La suite $(x_n)_{n\geq 0}$ est strictement décroissante puisque $x_n = f(x_{n-1}) < x_{n-1}$ pour tout $n \geq 1$. ♦

Exemple 8.4
L'ordre naturel est bien fondé sur $\mathbb{N}$, mais pas sur $\mathbb{Z}$, ni sur $\mathbb{Q}$, ni sur $\mathbb{R}$.

L'ordre $\sqsubseteq$ suivant est bien fondé sur $\mathbb{Z}$:
- Pour tout $n > 0$ et $m > 0$, $n \sqsubset m$ ssi $n < m$ ($\sqsubseteq$ et $\leq$ coïncident sur $\mathbb{N}$)
- Pour tout $n < 0$ et $m \geq 0$, $n \sqsubset m$ (les entiers négatifs sont plus petits que les entiers positifs)
- Pour tout $n < 0$ et $m < 0$, $n \sqsubset m$ ssi $m < n$ (ordre inverse sur les entiers négatifs)

L'ordre cartésien sur $\mathbb{N}^2$ est bien fondé. Plus généralement, *le produit de deux ordres bien fondés est bien fondé.*

Soit $\Sigma = \{a, b, \ldots\}$ (au moins deux lettres). L'ordre lexicographique n'est pas bien fondé : la suite $(a^n b)_{n\geq 0}$ est infinie et strictement décroissante.

L'ordre préfixe : pour tout $u, v \in \Sigma^*$, $u <_p v$ ssi il existe $w \in \Sigma^*$ tq $v = u.w$, est bien fondé sur Σ^*.

L'ordre : $u < v$ ssi $|u| < |v|$ ou $|u| = |v|$ et $u <_p v$ est bien fondé sur Σ^*. ♦

8.1.2 Suites décroissantes d'environnements

L'application à la terminaison des programmes consiste à définir un ordre bien fondé sur l'ensemble des environnements du programme et à identifier un terme dont les variables libres sont des variables du programme, nommé *variant* ou *mesure*, qui est strictement décroissant pour cet ordre.

Exemple 8.5

r := 0;
while $(n \geq (r+1)^2)$ **do**
  r := $r+1$
end

Une étude de cet algorithme montre que le terme $n - r^2$ est strictement décroissant et borné. On le choisi comme mesure.

Pour ce programme, un environnement est une fonction $v : \{r, n\} \rightarrow \mathbb{N}$. On considère l'ordre suivant :

$$v \sqsubset w \quad \text{ssi} \quad [n - r^2]_v < [n - r^2]_w$$

Nous avons alors :
— $n - r^2$ est strictement décroissant pour $\sqsubset$ sur toute exécution car la valeur de r augmente alors que celle de n est inchangée ;
— l'ordre $\sqsubset$ est bien fondé sur l'ensemble des environnements du programme car $n - r^2$ est à valeur dans $\mathbb{N} \cup \{n - k, \ldots, -1\}$ où k est le plus petit carré parfait strictement supérieur à n.

On voit plus loin comment intégrer la preuve de terminaison dans le calcul de Hoare pour obtenir une preuve formelle de terminaison.

8.1.3 Limites de l'approche

On rappelle tout d'abord que le problème de terminaison des while-programs est indécidable. Cette méthode est donc nécessairement incomplète.

Cette approche très simple permet de prouver la terminaison de nombreux programmes, et ne permet pas de prouver la terminaison de programmes encore plus nombreux. Il n'est pas rare de ne pas disposer d'une mesure qui :
— est à valeur dans un ensemble bien fondé : exemple des programmes manipulant des réels ou des flottants, fréquents dans l'industrie ;
— est strictement décroissante à *chaque* itération de la boucle : exemple, décroissance stricte une itération sur deux.

Il faut alors étudier les suites d'environnements générées par le programme pour comprendre comment en extraire une suite satisfaisante, lorsqu'elle existe (voir TD).

8.2 Correction : invariants de boucle

8.2.1 Extension du calcul de Hoare pour `while`

Le nombre d'itérations d'un while-program et les valeurs de variables à chaque itération ne sont généralement pas connus a priori : cela dépend des paramètres du

programme. Cependant, la preuve est *indépendante des paramètres*. C'est à dire qu'elle doit être valide quelle que soit la valeur de chaque paramètre.

La preuve de correction d'une boucle `while` repose donc sur un *invariant* de boucle. C'est à dire, une propriété qui est vraie à chaque itération de la boucle. L'objectif de la preuve est de démontrer que cet invariant est vrai à la sortie de la boucle `while`. Pour cela, on réalise une preuve par induction :

— on montre que l'invariant est vrai avant d'entrer dans la boucle ;
— et que si l'invariant est vrai avant d'exécuter le corps de la boucle, alors il est vrai après l'exécution du corps de la boucle (invariance).

On peut alors en déduire que l'invariant est vrai à chaque itération de la boucle, et donc, en particulier, lorsque l'on sort de la boucle.

Exemple 8.6
On considère à nouveau l'algorithme de calcul de la partie entière de la racine carrée de n. On note W l'instruction `while`.

```
r := 0;
while (n ≥ (r + 1)²) do
  r := r + 1
end
```

Exécutions et valeur des variables en fonction de la valeur de n :

$$n = 2 : \qquad (\texttt{r} := \texttt{0}; W, 2, -) \to (W, 2, 0) \to (\texttt{r} := \texttt{r} + \texttt{1}; W, 2, 0) \to (W, 2, 1) \to (\emptyset, 2, 1)$$

$$n = 5 : \qquad (\texttt{r} := \texttt{0}; W, 5, -) \to (W, 5, 0) \to (\texttt{r} := \texttt{r} + \texttt{1}; W, 5, 0) \to (W, 5, 1) \to$$
$$(\texttt{r} := \texttt{r} + \texttt{1}; W, 5, 1) \to (W, 5, 2) \to (\emptyset, 5, 2)$$

On peut remarquer que la propriété $r^2 \leq n$ est satisfaite dans tous les états de la forme $(W, \dots)$, ainsi que dans l'état $(\emptyset, \dots)$. Nous allons montrer qu'il s'agit d'un invariant de boucle :

(Base) $r^2 \leq n$ est vraie avant d'entrer dans la boucle car $n \in \mathbb{N}$ et $r = 0$ (après l'instruction `r := 0`.

(Induction) si $r^2 \leq n$ est vraie en entrée de boucle, elle est également vraie après l'exécution du corps de la boucle `r := r+1`. En effet, puisqu'on rentre la boucle, on a $n \geq (r+1)^2$. Ceci implique $n \geq r^2$ après exécution de `r := r+1`, c'est à dire $r^2 \leq n$, notre invariant.

On en déduit donc que $r^2 \leq n$ est vraie en sortie de boucle. Sachant que le test de la boucle est faux en sortie de boucle, on a $r^2 \leq n < (r+1)^2$ à la fin du programme, et donc r est bien la partie entière de $\sqrt{n}$ (c'est à dire la racine par défaut de n). ♦

Définition 8.7 Un triplet $\{P\}\ S\ \{R\}$ est *prouvable* s'il est construit par l'application un nombre fini de fois des règles de la définition 7.3 et de la règle ci-dessous :

6. while : $\dfrac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\ \{I \wedge \neg B\}}$

Dans cette règle, I est l'*invariant* de boucle. La prémisse de la règle demande de prouver que I est invariante par S, le corps de la boucle. ♦

Exemple 8.8
Preuve du programme de calcul de la racine par défaut. On prend $r^2 \leq n$ comme invariant.

La preuve est donnée en figure 8.1. Il reste à prouver la validité des trois *obligations de preuve* :

— $n \in \mathbb{N} \implies 0^2 \leq n$ par définition de $\mathbb{N}$ et $0^2 = 0$.
— $r^2 \leq n \wedge n \geq (r+1)^2 \implies (r+1)^2 \leq n$ évident syntaxiquement
— $r^2 \leq n \wedge \neg(n \geq (r+1)^2) \implies r^2 \leq n < (r+1)^2$ par $\neg(n \geq (r+1)^2)$ il vient $n < (r+1)^2$, le reste est syntaxique.

On peut conclure que le triplet $\{n \in \mathbb{N}\}\ \mathtt{r} := 0; W\ \{r^2 \leq n < (r+1)^2\}$ est valide. Donc cet algorithme calcule la racine carrée par défaut r de tout entier naturel n. ♦

Le théorème 7.5 s'étend à la règle `while` :

Théorème 8.9 (Complétude et correction) Le calcul de Hoare incluant la règle de l'instruction `while` est correct et relativement complet. ♦

8.2.2 Comment trouver des invariants de boucle ?

La difficulté de la preuve de programme est de trouver pour chaque boucle un invariant qui permet de conclure la preuve de correction. Il n'y a pas de méthode générale pour cela. On donne ici des principes qu'il peut être utile de suivre.

Ces deux principes s'appuient sur le fait que l'obligation de preuve correspondant à la post-condition de `while` doit être valide.

Utilisation de la postcondition de la règle `while`

Exemple 8.10
Racine carrée par défaut :

```
{n ∈ ℕ}
r := 0;
while (n ≥ (r+1)²) do
   r := r+1
end
{r² ≤ n ∧ n < (r+1)²}
```

On cherche I tel que :

$$I \wedge \neg(n \geq (r+1)^2) \quad \implies \quad r^2 \leq n < (r+1)^2$$

On prend $I \quad = \quad r^2 \leq n$. ♦

$$
\dfrac{
\dfrac{n \in \mathbb{N} \implies 0^2 \leq n \qquad \dfrac{}{\{0^2 \leq n\}\, \mathtt{r := 0}\, \{r^2 \leq n\}}}{\{n \in \mathbb{N}\}\, \mathtt{r := 0}\, \{r^2 \leq n\}}
\qquad
\dfrac{
\dfrac{
\dfrac{(r^2 \leq n \,\wedge\, n \geq (r+1)^2) \implies (r+1)^2 \leq n \qquad \dfrac{}{\{(r+1)^2 \leq n\}\, r := r+1\, \{r^2 \leq n\}}}{\{r^2 \leq n \,\wedge\, n \geq (r+1)^2\}\, r := r+1\, \{r^2 \leq n\}}
}{\{r^2 \leq n\}\, W\, \{r^2 \leq n \wedge \neg(n \geq (r+1)^2)\}}
\qquad (r^2 \leq n \wedge \neg(n \geq (r+1)^2)) \implies (r^2 \leq n < (r+1)^2)
}{\{r^2 \leq n\}\, W\, \{r^2 \leq n < (r+1)^2\}}
}{\{n \in \mathbb{N}\}\, \mathtt{r := 0};\, W\, \{r^2 \leq n < (r+1)^2\}}
$$

FIGURE 8.1 – Preuve de l'algorithme de calcul de la racine carrée par défaut avec l'invariant $r^2 \leq n$.

Exemple 8.11
Calcul de la factorielle :

$\{n \geq 0\}$
i := 0; f := 1;
while $(i \neq n)$ **do**
  i := $i+1$; f := $f \times i$
end
$\{f = \prod_{j=1}^{n} j\}$

On cherche I tel que :

$$I \wedge \neg(i \neq n) \quad \Longrightarrow \quad f = \prod_{j=1}^{n} j$$

Solution, puisque $i = n$ à la sortie de boucle, substituer i à n, ce qui donc : $I = f = \prod_{j=1}^{i} j$. On remarque que l'invariant capture bien l'accumulation realisée par la boucle pour le calcul de la factorielle. ♦

8.2.3 Intégration de la preuve de terminaison

La règle de `while` peut être modifiée pour prouver à la fois la terminaison et la correction.

Dans l'exemple ci-dessus, l'ordre $\sqsubset$ est définit relativement au terme $n-r^2$. Ce terme est appelé *mesure* (*variant* en anglais). La première étape de la preuve consiste donc à identifier pour chaque boucle une mesure que l'on pense :
— strictement décroissante par exécution du corps de la boucle ;
— à valeur dans un ensemble bien fondé.

Une fois un *variant* (ou *mesure*) identifié, on doit prouver qu'il est strictement décroissant et à valeur dans un ensemble bien fondé. On remplace la règle de la définition 8.7 par :

$$\frac{< \text{bien fondé} \qquad \{I \wedge B \wedge (m = X)\}\ S\ \{I \wedge (m < X)\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\ \{I \wedge \neg B\}}$$

où m désigne la mesure et X est un nom de variable qui n'apparaît ni dans le programme ni dans les annotations.

Intuitivement, X conserve la valeur de m avant l'exécution de S. En effet, X n'est pas modifiée par le programme, donc $m < X$ après l'exécution de S prouve que la valeur de m est strictement décroissante par exécution de S.

Année 2018/2019 TD Logique propositionnelle

LOGIQUE ET PREUVE DE PROGRAMMES
IF107

Filière : Informatique, Année : 1
Web: `http://herbrete.vvv.enseirb-matmeca.fr/IF107`

Exercice 1 (Implication)

Répondre aux questions suivantes en vous basant sur l'affirmation "les personnes qui ont des chats ont également des chiens".

1. Si une personne n'a pas de chien, peut-on en déduire qu'elle n'a pas de chat ?
2. Si une personne n'a pas de chat, peut-on en déduire qu'elle n'a pas de chien ?
3. Les phrases suivantes sont-elles équivalentes à l'affirmation ci-dessus ? Lesquelles sont équivalentes entre elles ?
   (a) "les personnes qui n'ont pas de chat, n'ont pas non plus de chiens"
   (b) "les personnes qui n'ont pas de chien, n'ont pas non plus de chat"
   (c) "les personnes qui ont des chiens ont également des chats"
4. Traduire ces 4 affirmations en logique propositionnelle en utilisant les proprositions `a-chat` et `a-chien`

Exercice 2 (Tables de vérité)

Pour les formules suivantes :
— introduire les parenthèses en respectant la priorité des opérateurs,
— calculer la table de vérité et indiquer si la formule est satisfaisable et si elle est valide.

1. $p \wedge (q \vee r)$
2. $p \implies (q \implies p)$
3. $p \wedge \neg q \wedge r$
4. $(p \vee q) \wedge (\neg p \vee r) \wedge (\neg q \vee p) \wedge (\neg q \vee r) \wedge (\neg p \vee \neg r)$

Exercice 3 (Énigme)

Vous êtes perdu sur une piste dans le désert. Vous arrivez à une bifurcation. Chacune des deux pistes est gardée par un sphinx que vous pouvez interroger. Les pistes peuvent soit conduire à une oasis, soit se perdre dans un désert profond (au mieux, elles conduisent toutes à une oasis, au pire elles se perdent toutes les deux). Vous disposez des informations suivantes :

A. Le sphinx de droite vous répond : "Une au moins des deux pistes conduit à une oasis".

B. Le sphinx de gauche vous répond : "La piste de droite se perd dans le désert".

C. Vous savez que les sphinx disent tous les deux la vérité, ou bien mentent tous les deux.

1. Soit D est la proposition "Il y a une oasis au bout de la route droite" et soit G est la proposition "Il y a une oasis au bout de la route gauche". Exprimer les affirmations A, B et C en logique propositionnelle (formules ϕ_A, ϕ_B, ϕ_C).
2. Construire la table de vérité de formules ϕ_A, ϕ_B et ϕ_C.
3. Répondre aux questions suivantes en justifiant votre raisonnement par la table de vérité.
   (a) peut-il y avoir deux oasis ?
   (b) les sphinx ont-ils menti tous les deux ?
   (c) peut-il avoir une oasis à doite ?
   (d) y-a-t'il une oasis à gauche ?

Exercice 4 (Équivalences logiques)

1. Démontrer les équivalences suivantes :

$$
\begin{aligned}
A \vee (B \wedge C) &\equiv (A \vee B) \wedge (A \vee C) \\
A \wedge (B \vee C) &\equiv (A \wedge B) \vee (A \wedge C) \\
\neg(A \wedge B) &\equiv \neg A \vee \neg B \\
\neg(A \vee B) &\equiv \neg A \wedge \neg B \\
A \implies B &\equiv \neg A \vee B
\end{aligned}
$$

2. Simplifier les formules $(A \vee B) \wedge B$, $(A \wedge B) \vee A$
3. Donner une formule équivalente à $A \implies B \implies C$ qui n'utilise pas l'implication

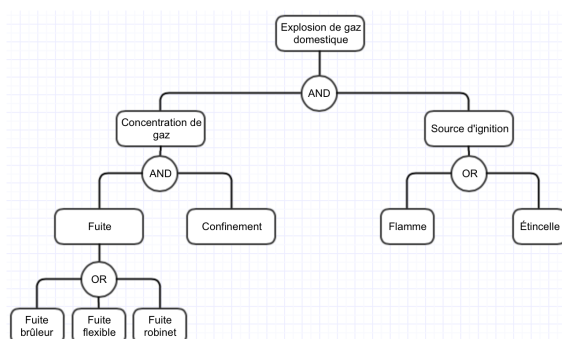
Exercice 5 **(Fini)**

Soient deux variables propositionnelles p et q.

1. Montrer qu'il existe une infinité de formules utilisant ces 2 variables
2. Combien y-a-t'il de tables de vérité utilisant ces deux variables ?
3. Combien de formules construites sur n variables propositionnelles faut-il prendre pour en avoir deux différentes qui sont équivalentes ?

Exercice 6 **(Arbres de défaillances)**

En sûreté de fonctionnement, une analyse des risques est souvent conduite en construisant un arbre de défaillance. Par exemple :



L'arbre est construit en partant de l'événement redouté (ici, "explosion de gaz domestique") et en analysant les causes possibles jusqu'à obtenir des causes élémentaires. L'arbre représente in fine les combinaisons de ces causes pouvant conduire à la catastrophe.

1. Soit $P = \{a, b, c, d, e, f\}$ les variables propositionnelles associées aux feuilles de l'arbre. Donner la formule correspondant à l'arbre ci-dessus.
2. La sûreté de fonctionnement s'intéresse aux "coupes minimales", c'est à dire aux plus petits ensembles de causes élémentaires pouvant produire l'événement redouté. Proposer un algorithme simple pour calculer les coupes minimales d'une formule donnée.
3. La sûreté de fonctionnement considère qu'un système critique est sûr si ses coupes minimales sont de taille au moins 3. L'installation de gaz domestique ci-dessus est-elle sûre ?

NB : les arbres de défaillances sont utilisées dans l'industrie (aéronautique, nucléaire, etc) pour les analyses de défaillances et le contrôle des risques industriels. ♦

Exercice 7 **(Spécification et résolution d'un jeu)**

	1	2	3
1			
2	**3**		
3			

On considère un jeu proche du Sudoku représenté ci-contre. Une solution du jeu attribue à chaque case de la grille un chiffre parmi $\{1, 2, 3\}$ tel que chaque chiffre apparaît de manière unique dans chaque ligne, chaque colonne et chaque bloc coloré.

Le but de cet exercice est de modéliser les solutions du jeu par une formule de *logique propositionnelle* ϕ qui est *satisfaisable si et seulement si le jeu admet une solution*. Comme en cours, nous utilisons pour cela un ensemble de variables propositionnelles p_{ijk}, avec la signification suivante : p_{ijk} est vraie lorsque la case (i, j) contient la valeur k, et fausse sinon. Dans la grille ci-dessous, nous avons p_{213} puisque la case $(2, 1)$ contient la valeur 3 et $\neg p_{211}$ puisque la case $(2, 1)$ ne contient pas la valeur 1.

1. Écrire une formule de logique propositionnelle qui exprime ≪ s'il y a 1 en case $(1, 1)$, il n'y a pas 1 en case $(1, 2)$, ni en case $(1, 3)$ ≫.
2. Généraliser la formule précédente à : ≪ s'il y a k en case (i, j), alors il n'y a pas k dans les cases de l'ensemble C ≫. On écrira $\bigwedge_{(i',j')\in C} \cdots$ pour quantifier sur toutes les cases de C (c'est à dire pour écrire ≪ pour toute case (i', j') de C . . . ≫). *Attention : (i, j) peut appartenir à C.*
3. On note $\phi_{i,j,k,C}$ la formule obtenue à la question précédente. À l'aide de $\phi_{i,j,k,C}$ écrire une formule qui exprime : ≪ la valeur k apparaît au plus une fois dans l'ensemble de cases C ≫.
4. On note $\phi_{k,C}$ la formule obtenue à la question précédente. À l'aide de $\phi_{k,C}$ écrire une formule qui exprime : ≪ chaque valeur $k \in \{1, 2, 3\}$ apparaît au plus une fois dans l'ensemble de cases C ≫.
5. On note ϕ_C la formule obtenue à la question précédente. Écrire une formule qui exprime les contraintes d'unicité sur les lignes, les colonnes et les blocs de la grille en utilisant ϕ_C. On notera L_1, L_2 et L_3 les ensembles de cases correspondant aux lignes, C_1, C_2 et C_3 ceux correspondant aux colonnes, et B_1, B_2 et B_3 ceux correspondant aux blocs colorés.
6. Quelle(s) autre(s) contrainte(s) faut-il ajouter à la formule précédente pour obtenir une formule ϕ qui est satisfaisable si et seulement si la grille admet une solution ? Justifier.
7. On fixe usuellement la valeur de certaines cases, comme la case $(2, 1)$ dans l'exemple ci-dessus pour restreindre les solutions possibles. Comment ces valeurs fixées peuvent-elles être modélisées dans la formule ϕ ?
8. Décrire en 2 à 3 lignes un algorithme pour déterminer si le jeu admet une solution à partir de la formule ϕ.
9. Supposons qu'un algorithme fournit une solution s au jeu à partir de ϕ. Comment utiliser cet algorithme pour vérifier si s est l'unique solution du jeu ?
10. Supposons connue une solution du jeu. Donner un algorithme qui permet de construire une grille à trous qui possède une unique solution.

Année 2018/2019 TD Logique des prédicats

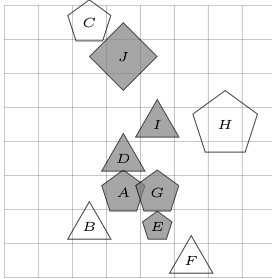
LOGIQUE ET PREUVE DE PROGRAMMES
IF107

Filière : Informatique, Année : 1

Web: `http://herbrete.vvv.enseirb-matmeca.fr/IF107`

Exercice 1 (Évaluation de formules)

Soit l'ensemble de figures représentées sur la grille ci-dessous. Le pentagone E est une *petite* figure. Les figures H et J sont de *grandes* figures. Toutes les autres sont des figures de taille *moyenne*. Indiquez pour chaque formule ci-dessous si elle est vraie pour x et y interprétées sur l'ensemble des figures ci-dessous. Par exemple $\mathsf{pluspetite}(x, y)$ se lit "x est une figure strictement plus petite que y".



1. $\mathsf{triangle}(C)$
2. $\mathsf{pentagone}(F) \implies \mathsf{cercle}(H)$
3. $\exists x.\, \neg\, \mathsf{triangle}(x)$
4. $\forall x. \exists y.\, \mathsf{pluspetite}(x, y)$
5. $\forall x. \exists y.\, \mathsf{a_gauche}(x, y)$
6. $\exists x. \forall y.\, (y \neq x \implies \mathsf{a_droite}(x, y))$
7. $\exists x. \forall y.\, \neg\, \mathsf{pluspetite}(x, y)$
8. $\forall x.\, (\mathsf{pentagone}(x) \implies \exists y.\, (\mathsf{triangle}(y) \wedge \mathsf{a_droite}(x, y)))$

Exercice 2 (Quantification)

1. Écrire une formule équivalente à $\exists x.\, p(x)$ sans quantificateur pour x à valeur dans $\{a, b, c\}$. Faire de même avec la formule $\forall .x\, p(x)$.
2. On considère maintenant que x est à valeurs dans un ensemble infini. Quel est le rôle des quantificateurs ?

Exercice 3 (Formaliser)

On souhaite modéliser différentes contraintes pour la réalisation d'équipes pour un tournoi sportif. On se donne les prédicats suivants :

meme_club(x,y)	x et y sont licenciés dans le même club sportif
equipe(x, y)	x et y appartienent à la même équipe
femme(x)	si x est une femme
$x = y$	si x et y sont la même personne
$x \neq y$	si x et y sont deux personnes distinctes

Exprimer les contraintes suivantes en logique des prédicats :

1. La relation `equipe` est (a) symmétrique, (b) transitive et (c) irréflexive
2. Tout joueur est l'équipier d'au moins un autre joueur
3. Les équipes sont constitués de joueurs du même club
4. Toute équipe est mixte
5. Un équipe est constituée d'au moins trois joueurs

Exercice 4 (Requêtes)

On considère la base de données constituée des relations suivantes :

— produit($idp, description$)
— client(idc, nom)
— vente(idp, idc)

Exprimer les requêtes suivantes en logique des prédicats en utilisant les relations ci-dessus.

1. nom_client(nom) qui est satisfaite si nom est un nom de client
2. ventes_shampooing(nom) qui est satisfaite si nom est le nom d'un client qui a acheté du shampooing
3. invendu($description$) qui est satisfaite si $description$ est un produit invendu

Exercice 5 (Satisfaisabilité et validité)

Pour chaque formule ci-dessous, donner sa structure, ses occurrences libres et ses occurrences liées. Indiquer celles qui sont satisfaisables, valides ou insatisfaisables.

1. $(\forall x.\neg p(x)) \vee (\exists x.p(x))$
2. $p(x) \implies (\forall x.p(x))$
3. $(\exists x.p(x)) \implies p(z)$
4. $\forall x.\forall y.(p(x) \iff \neg p(y))$
5. $(\forall x.\exists y.p(x,y)) \implies (\exists y.\forall x.p(x,y))$
6. $(\forall x.p(x) \wedge q(x)) \iff (\forall x.\, p(x)) \wedge (\forall x.q(x))$
7. $(\forall x.p(x)) \vee (\forall x.q(x)) \implies (\forall x.p(x) \vee q(x))$
8. $(\forall x.p(x) \vee q(x)) \implies (\forall x.p(x)) \vee (\forall x\, q(x))$
9. $(\exists x.p(x) \wedge q(x)) \implies (\exists x.p(x)) \wedge (\exists x.q(x))$
10. $(\exists x.p(x)) \wedge (\exists x.q(x)) \implies \exists x.(p(x) \wedge q(x))$
11. $(\exists x.p(x) \vee q(x)) \iff (\exists x.p(x)) \vee (\exists x.q(x))$

	Année 2018/2019	TD Preuve formelle
	Logique et preuve de programmes IF107	
	Filière : Informatique, Année : 1	
	Web: `http://herbrete.vvv.enseirb-matmeca.fr/IF107`	

1 Un calcul simple

Soit N l'ensemble des mots constitués d'une séquence non vide de symboles $\bullet$. Intuitivement N représente les entiers naturels *non nuls* en base unaire : $\bullet$ représente 1, $\bullet\bullet$ représente 2, $\bullet\bullet\bullet$ représente 3, etc. Dans la suite, x et y représentent deux mots de N. On note $n(x)$ et $n(y)$ les nombres qu'ils représentent (i.e. leur longueur). Leur concaténation xy représente donc la somme : $n(xy) = n(x) + n(y)$.

Exercice 1 (La règle NDP)

On considère dans un premier temps le système formel constitué de l'axiome (Ax) et de la règle de déduction (R_1) ci-dessous, où la proposition "$x\,NDP\,y$" se lit "x Ne Divise Pas y".

$$\frac{}{xy\,NDP\,x}\ (Ax) \qquad\qquad \frac{x\,NDP\,y}{x\,NDP\,xy}\ (R_1)$$

1. Prouver la proposition $\bullet\bullet\ NDP\ \bullet\bullet\bullet$ dans ce système formel
2. Que se passe-t-il si on tente de prouver une proposition invalide comme $\bullet\bullet\ NDP\ \bullet\bullet\bullet\bullet$?
3. Montrer *par induction*[1] que $x\,NDP\,y$ est prouvable si et seulement si $n(x)$ ne divise pas $n(y)$.

Exercice 2 (La règle SD)

On ajoute maintenant les règles (R_2) et (R_3) ci-dessous au système formel précédent, où la proposition "$x\,SD\,y$" se lit "x est Sans Diviseur entre 2 (i.e. $\bullet\bullet$) et y".

$$\frac{\bullet\bullet\,NDP\,x}{x\,SD\,\bullet\bullet}\ (R_2) \qquad\qquad \frac{x\bullet\,NDP\,y \qquad y\,SD\,x}{y\,SD\,x\bullet}\ (R_3)$$

1. Donner une preuve qui montre la validité de $\bullet\bullet\bullet\bullet\bullet\ SD\ \bullet\bullet\bullet$
2. Montrer *par induction* que si $x\,SD\,y$ est prouvable, alors $n(x)$ n'admet aucun diviseur entre 2 et $n(y)$.

1. La preuve par induction d'une propriété P pour une règle R consiste à montrer que P est vraie pour la conclusion de R en supposant que P est vraie pour les prémisses de R.

Exercice 3 **(La règle P)**

On ajoute finalement l'axiome (Ax) et la règle (R_4) ci-dessous au système formel précédent, où la proposition "$P\,x$" se lit "x est Premier".

$\frac{}{P\,\bullet\bullet}\quad (Ax_2)$ $\qquad\qquad$ $\frac{x\bullet\,SD\,x}{P\,x\bullet}\quad (R_4)$

1. Donner une preuve qui montre la validité de $P\ \bullet\bullet\bullet$.
2. Montrer *par induction* que si $P\,x$ est prouvable, alors $n(x)$ est un nombre premier.

Cette séquence d'exercices (tirée du sujet d'examen 2010/2011) est inspirée de "Gödel, Escher et Bach, les brins d'une guirlande éternelle", D. Hofstadter, Inter éditions, 1993.

2 Calcul des séquents propositionnel

1. utilisation d'une hypothèse : $\frac{}{\Gamma, \psi \vdash \psi}$
2. augmentation des hypothèses : $\frac{\Gamma \vdash \phi \quad \psi \notin \Gamma}{\Gamma, \psi \vdash \phi}$
3. détachement (Modus ponens) : $\frac{\Gamma \vdash (\phi \implies \phi') \quad \Gamma \vdash \phi}{\Gamma \vdash \phi'}$
4. retrait d'une hypothèse : $\frac{\Gamma, \psi \vdash \phi}{\Gamma \vdash (\psi \implies \phi)}$
5. double négation : $\frac{\Gamma \vdash \phi}{\Gamma \vdash \neg\neg\phi}$ $\qquad$ $\frac{\Gamma \vdash \neg\neg\phi}{\Gamma \vdash \phi}$
6. contradiction : $\frac{\Gamma, \psi \vdash \phi \qquad \Gamma, \psi \vdash \neg\phi}{\Gamma \vdash \neg\psi}$
7. conjonction : $\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \phi'}{\Gamma \vdash (\phi \wedge \phi')}$ $\qquad$ $\frac{\Gamma \vdash (\phi \wedge \phi')}{\Gamma \vdash \phi}$ $\qquad$ $\frac{\Gamma \vdash (\phi \wedge \phi')}{\Gamma \vdash \phi'}$
8. disjonction : $\frac{\Gamma, \psi \vdash \phi \qquad \Gamma, \neg\psi \vdash \phi'}{\Gamma \vdash (\phi \vee \phi')}$ $\qquad$ $\frac{\Gamma, \psi \vdash \phi \qquad \Gamma, \psi' \vdash \phi}{\Gamma, (\psi \vee \psi') \vdash \phi}$

Exercice 4 (Premières preuves)

Montrer que les séquents suivants sont prouvables. Pour chacun des séquents, indiquer quelle formule est prouvée valide.

1. $A, B \vdash (A \wedge B)$
2. $A, (A \implies B) \vdash B$
3. $A, B, (A \implies B \implies C) \vdash C$
4. $(A \vee B), (A \implies C), (B \implies C) \vdash C$

♦

Exercice 5 (Lemmes)

Démontrer les règles suivantes à l'aide du calcul des séquents propositionnel.

1. $\dfrac{\Gamma, \psi, \psi' \vdash \phi}{\Gamma, (\psi \wedge \psi') \vdash \phi}$
2. $\dfrac{\Gamma, (\psi \wedge \psi') \vdash \phi}{\Gamma, \psi, \psi' \vdash \phi}$
3. Modus tollens : $\dfrac{\Gamma \vdash \neg\phi' \quad \Gamma \vdash (\phi \implies \phi')}{\Gamma \vdash \neg\phi}$

♦

Exercice 6 **(Equivalences logiques standards)**

Définir les séquents à prouver pour démontrer chacun des équivalences logiques ci-dessous. Donner une preuve pour chacun de ces séquents.

1. $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
2. $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
3. $\neg(A \vee B) \equiv \neg A \wedge \neg B$
4. $\neg(A \wedge B) \equiv \neg A \vee \neg B$
5. $A \Rightarrow B \equiv \neg A \Rightarrow \neg B$
6. $A \implies B \equiv \neg A \vee B$

♦

3 Calcul des séquents en logique des prédicats

Régles du calcul propositionnel, plus les règles ci-dessous pour les quantificateurs :

9. introduction : $\dfrac{\Gamma \vdash \phi}{\Gamma \vdash \forall x.\phi}$ **avec** x **non libre dans** Γ $\qquad \dfrac{\Gamma \vdash \phi[x \leftarrow t]}{\Gamma \vdash \exists x.\phi}$
10. élimination : $\dfrac{\Gamma \vdash \forall x.\phi}{\Gamma \vdash \phi[x \leftarrow t]}$ $\qquad \dfrac{\Gamma \vdash \exists x.\phi \quad \Gamma, \phi \vdash \psi}{\Gamma \vdash \psi}$ **avec** x **non libre dans** Γ **et** Ψ

où t est un terme quelconque (une valeur quelconque de x) et $\phi[x \leftarrow t]$ est la substitution de t à x dans ϕ.

Exercice 7 **(Preuve en séquents des prédicats)**

Soit R un symbole de relation binaire.

1. formaliser en logique des prédicats : R est totale, R est symmétrique, R est transitive et R est réflexive
2. formaliser par un séquent la proposition suivante : ”si R est totale, symmétrique et transitive, alors R est réflexive”
3. pouver le séquent ci-dessus

♦

Année 2018/2019 — TD Induction

LOGIQUE ET PREUVE DE PROGRAMMES
IF107

Filière : Informatique, Année : 1

Web: `http://herbrete.vvv.enseirb-matmeca.fr/IF107`

Exercice 1 (Arbres binaires)

Un arbre binaire x sur un alphabet Σ est soit l'arbre vide, noté nil, soit l'arbre $a(g,d)$ de racine $a \in \Sigma$, de fils gauche g et de fils droit d, où g et d sont des arbres binaires.

1. Donner une définition inductive de la fonction n qui à tout arbre binaire x associe le nombre de nœuds dans x.
2. Definir la fonction inductive nbv qui à tout arbre binaire x associe le nombre de sous-arbres vides dans l'arbre x.
3. Puis, prouver par induction que le nombre de sous-arbres vides dans un arbre à $n \geq 0$ nœuds est égal à $n+1$. Formellement, le prédicat à prouver $P(x)$ est défini pour tout arbre binaire x par $nbv(x) = n(x) + 1$.

Exercice 2 (Un jeu de piles)

Le débute avec une pile de K boîtes. À chaque tour de jeu, on divise une pile de hauteur $n_1 + n_2$ en deux piles de hauteurs n_1 et n_2 telles que $n_1, n_2 \geq 1$. Cette division rapporte $n_1 n_2$ points. Le jeu s'arrête lorsque l'on a K piles de hauteur 1. Le score est alors la somme des points obtenus à chaque division.

Démontrer par induction que le score final est $K(K-1)/2$ quelle que soit la stratégie utilisée.

Exercice 3 (Palindromes)

Un palindrome est un mot qui se lit de la même façon de gauche à droite et de droite à gauche, par exemple : laval, radar, été, ressasser,.... Dans la suite, on note L l'ensemble des palindromes sur l'alphabet $\{0,1\}$.

1. Quels sont les mots palindromes de longueur 2, et ceux de longueur 3 ?
2. En déduire une définition inductive de L et démontrer qu'elle est correcte, c'est à dire que tout palindrome est généré par votre définition, et que tout mot généré est un plaindrome.
3. Donner une expression du nombre de palindromes de longueur n. Quel est le nombre de palindromes de longueur $n = 351$? Généraliser à un alphabet à k symboles.

Exercice 4 (Une preuve fausse)

Un chercheur (fou) est parvenu à montrer le théorème suivant : "tous les chevaux ont la même couleur". Il a proposé la preuve par induction sur le nombre de chevaux retranscrite ci-dessous.

(Base) Dans un groupe ne contenant qu'un seul cheval, tous les chevaux ont la même couleur.

(Induction) Supposons le théorème vrai pour N chevaux. Considérons un groupe G de $N+1$ chevaux. Observons qu'en retirant un cheval quelconque de G, on obtient un groupe de N chevaux qui ont donc tous la même couleur par hypothèse d'induction. De même, en retirant maintenant *un autre* cheval de G, on obtient a nouveau un groupe de N chevaux qui ont tous la même couleur. On en déduit alors que tous les chevaux ont la même couleur dans le groupe de $N+1$ chevaux.

Cette preuve est évidemment fausse. Où se trouve l'erreur ? ⧫

Exercice 5 (Preuve d'algorithme récursif)

La fonction `length` suivante calcule la longueur d'une liste, le nombre d'éléments qu'elle contient. Une liste est soit la liste vide `[]`, soit une liste non vide `hd::tl` constituée d'un élément de tête `hd` et d'une liste `tl`.

```
1   length(l) = match l with
2                 [] -> 0
3               | hd::tl -> length(tl)+1
```

Prouver la correction de la fonction `length` par induction. ⧫

Exercice 6 (Preuve d'algorithme récursif (2))

La fonction `find` suivante indique si un élément `x` apparient à une liste. Une liste est soit la liste vide `[]`, soit une liste non vide `hd::tl` constituée d'un élément de tête `hd` et d'une liste `tl`.

```
1   find(x,l) = match l with
2                 [] -> false
3               | hd::tl -> if (hd=x) then
4                             true
5                           else
6                             find(x,tl)
```

Prouver la correction de la fonction `find` par induction. ⧫

Exercice 7 (TAD et induction)

Une file FIFO est une structure de données qui stocke une séquence d'éléments, et telle que l'ajout se fait en fin de séquence et le retrait se fait en tête de séquence [1]. L'ensemble des files (FIFO) est inductivement défini par :

(Base) `empty` est une file

(Induction) pour toute file `f` et pour tout élément `x`, `push(f,x)` est une file

Par exemple, $\texttt{push}(\texttt{push}(\texttt{empty}, \texttt{a}), \texttt{b})$ est la file FIFO correspondant à la séquence d'éléments ab. La fonction θ ci-dessous associe à une file `f` la séquence correspondante.

$$\theta(\texttt{empty}) = \varepsilon$$
$$\theta(\texttt{push}(\texttt{f}, \texttt{x})) = \theta(\texttt{f}) \cdot x$$

La séquence vide est notée ε et "$\cdot$" représente la concaténation. Ainsi :

$$\begin{aligned}\theta(\texttt{push}(\texttt{push}(\texttt{empty}, \texttt{a}), \texttt{b})) &= \theta(\texttt{push}(\texttt{empty}, \texttt{a})) \cdot b \\ &= \theta(\texttt{empty}) \cdot a \cdot b \\ &= \varepsilon \cdot a \cdot b \\ &= ab\end{aligned}$$

1. Par quelle file la séquence abc est-elle représentée ? Plus généralement, par quelle file la séquence $x_1 \cdots x_n$ est-elle représentée ?
2. Donner une définition inductive de `top(f)` qui retourne l'élément de tête d'une file `f` supposée non vide. Démontrez par induction que votre définition est correcte, c'est à dire que $\texttt{top}(\texttt{f}) = x_1$ si et seulement si $\theta(\texttt{f}) = x_1 \cdots x_n$ avec $n \geq 1$.
3. Donnez une définition inductive de `pop(f)` qui retourne la file `f` privée de son élément de tête, `f` étant supposée non vide. Démontrez par induction que votre définition est correcte, c'est à dire que $\theta(\texttt{pop}(\texttt{f})) = x_2 \cdots x_n$ si et seulement si $\theta(\texttt{f}) = x_1 x_2 \cdots x_n$ avec $n \geq 1$ (remarque : la séquence $x_2 \cdots x_n$ peut être vide).
4. L'opération `reverse(f)`, qui inverse le contenu d'une file `f`, est définie par :

```
reverse(empty) = empty
reverse(f)     = push( reverse( pop(f) ), top(f) )   [f != empty]
```

   Démontrez par induction que la définition de `reverse` est correcte : $\theta(\texttt{reverse}(f)) = x_n \cdots x_1$ si et seulement si $\theta(\texttt{f}) = x_1 \cdots x_n$ (la séquence vide est obtenue pour $n = 0$).

♦

1. Une file FIFO fonctionne exactement comme une file d'attente à un guichet.

1 Spécification de programmes

Exercice 1 (La bonne spécification)

On considère une fonction :

```
int find(int * t, int n, int x)
```

qui cherche `x` dans le tableau `t` de taille `n`. La fonction `find` retourne la position i telle que $\mathtt{t}[i] = \mathtt{x}$ si `x` appartient à `t` et -1 sinon.

1. Pourquoi la spécification suivante ne correspond pas à celle de la fonction `find` (r est la valeur de retour de `find`) ?
   — PRECOND : $(n \in \mathbb{N})$
   — POSTCOND : $(r = -1) \vee (0 \leq r < n \wedge t[r] = x)$
2. Proposer une spécification correcte pour `find`.

Exercice 2 **(Spécification de programmes)**

Donner des pré-conditions et des post-conditions pour les programmes ci-dessous en logique du 1er ordre. On nommera `r` la valeur retournée par ces fonctions.

1. La fonction `max` qui retourne le maximum de `x` et de `y`

   ```
   int max(int x, int y);
   ```

2. La fonction `fact` calcule la factorielle de `n`

   ```
   int fact(int n);
   ```

3. La fonction `abs` calcule la valeur absolue de `x`

   ```
   int abs(int x);
   ```

4. La fonction `gcd` calcule le plus grand commun diviseur à `a` et `b`

   ```
   int gcd(int a, int b);
   ```

5. La fonction `xchange` échange les valeurs de `x` et `y`.

   ```
   void xchange(int * x, int * y);
   ```

6. La fonction `sort` qui trie le tableau `t` à `n` éléments (ordre croissant)

   ```
   void sort(int * t, int n);
   ```

7. La fonction `array_eq` qui indique si les tableaux `t1` et `t2` à `n` éléments sont identiques (convention usuelle : 0 pour faux et tout autre valeur pour vrai)

   ```
   int array_eq(int * t1, int * t2, int n);
   ```

2 Preuves de programmes

Dans tous les exercices suivants, on demande de donner une spécification de l'algorithme, puis d'en démontrer la terminaison (avec la méthodes des ensembles bien fondés) et la correction (par induction pour les programmes récursifs, et en prouvant en invariant pour les programmes itératifs).

Exercice 3 (Factorielle)

Les deux algorithmes ci-dessous calculent la factorielle de n.

```
1 fact(n):
2   i := 1; f := 1;
3   while (i ≤ n) do
4     f := f*i;
5     i := i+1
6   end;
7   return f
```

```
1 fact_rec(n):
2   if (n = 0) then
3     return 1;
4   else
5     return n*fact_rec(n-1)
```

Exercice 4 (Recherche linéaire)

Les algorithmes suivants recherchent x dans le tableau t de taille n. Ils retournent $true$ si x appartient à t et $false$ sinon.

```
1 find(t,n,x):
2   i := 0;
3   while ((i < n) ∧ (t[i] ≠ x)) do
4     i := i+1
5   end;
6   return (i < n)
```

```
1 find_rec(t,n,x):
2   if (n = 0) then
3     return false
4   else if (t[n - 1] = x) then
5     return true
6   else
7     return find_rec(t,n-1,x)
8   end
```

Exercice 5 (Recherche dichotomique)

Les fonctions suivantes recherchent x dans le tableau t de taille n trié par ordre croissant. Elles retournent $true$ si x appartient à t et $false$ sinon. Comparer l'invariant à celui de l'exercice précédent.

```
1  binfind(t,n,x):
2    l := 0; r := n−1;
3    p := (l + r) / 2;
4    while ((l ≤ r) ∧ (t[p] ≠ x)) do
5      if (t[p] < x) then
6        l := p + 1
7      else // (t[p] > x)
8        r := p − 1
9      end;
10     p := (l + r) / 2
11   end;
12   return (l ≤ r);
```

```
1  binfind_rec(t,l,r,x):
2    if (l > r) then
3      return false
4    else
5      p := (l + r) / 2;
6      if (t[p] = x) then
7        return true;
8      else if (t[p] < x) then
9        return binfind_rec(t,p+1,r,x)
10     else
11       return binfind_rec(r,l,p−1,x);
12     end
13   end
14
15 binfind(t,n,x):
16   return binfind_rec(t,0,n−1,x)
```

Exercice 6 **(Tri par extraction)**

1. Soit l'algorithme suivant :

```
1  selection(t, N, m):
2     imax := 0;  i := 1;
3     while (i ≤ m) do
4        if (t[i] > t[imax])
5           imax := i
6        end;
7        i := i + 1
8     end;
9     return imax
```

Cet algorithme retourne un indice $imax \in [0; m]$, avec $m < N$, tel que $t[imax]$ est un maximum de t.

(a) Écrivez en logique des prédicats, une pré-condition et une post-condition pour l'algorithme `selection`.

(b) Justifiez la terminaison de `selection` par la méthode des ensembles bien fondés.

(c) Donnez un invariant de boucle et justifiez la correction de `selection`.

On considère une fonction $\texttt{swap}(t, i, j, N)$ qui retourne un tableau t' identique à t sauf les valeurs d'indices i et j qui ont été permutées. Sa spécification est la suivante :

PRECOND : $(0 \leq i < N) \wedge (0 \leq j < N)$

POSTCOND : $(t'[i] = t[j]) \wedge (t'[j] = t[i]) \wedge (\forall k. (k \neq i \wedge k \neq j) \implies (t'[k] = t[k])$

2. Soit l'algorithme `sort` suivant :

```
1  sort(t, N):
2     i := N − 1;  imax := 0;
3     while (i ≥ 1) do
4        imax := selection(t, N, i);
5        t := swap(t, N, i, imax);
6        i := i − 1
7     end
```

La fonction $\texttt{sort}(t, N)$ trie le tableau t à N éléments par la méthode d'extraction du maximum.

(a) Écrivez en logique des prédicats, une pré-condition et une post-condition pour l'algorithme `sort`.

(b) Justifiez la terminaison de `sort` par la méthode des ensembles bien fondés.

(c) Donnez un invariant de boucle et justifiez la correction de `sort`.

♦

LOGIQUE ET PREUVE DE PROGRAMMES
IF107

Filière : Informatique, Année : 1
Web: `http://herbrete.vvv.enseirb-matmeca.fr/IF107`

Règles du calcul de Hoare pour les while-programs :

$$\frac{}{\{R\}\ \texttt{skip}\ \{R\}}\ (skip)$$

$$\frac{}{\{R[x \leftarrow t]\}\ \texttt{x := t}\ \{R\}}\ (aff.)$$

$$\frac{\{P\}S_1\{Q\} \quad \{Q\}S_2\{R\}}{\{P\}\ S_1; S_2\ \{R\}}\ (seq.)$$

$$\frac{\{P \wedge B\}\ S_1\ \{R\} \quad \{P \wedge \neg B\}\ S_2\ \{R\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end}\ \{R\}}\ (if)$$

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\ \{I \wedge \neg B\}}\ (while)$$

$$\frac{P' \implies P \quad \{P\}\ S\ \{R\} \quad R \implies R'}{\{P'\}\ S\ \{R'\}}\ (cons.)$$

Exercice 1 (Validité de triplets de Hoare)

Prouver la validité des triplets suivants à l'aide du calcul de Hoare.

1. $\{y > 0\}\ x := y\ \{x > 0\}$
2. $\{y \geq 10\}\ x := y\ \{x > 0\}$
3. $\{z < -3\}\ y := 3 * z + 7; x := 2 * y\ \{x < z\}$
4. $\{x = X \wedge y = Y\}\ z := x; x := y; y := z\ \{x = Y \wedge y = X\}$
5. $\{x = X\}$ **if** $(x < 0)$ **then** $x := -x$ **else skip end** $\{x = |X|\}$
6. Que se passe-t-il lorsqu'on inverse les instructions $x := -x$ et `skip` dans le triplet précédent ?

Exercice 2 (Nouvelle règle de while)

$$\frac{I \wedge B \implies P \quad \{P\}\,S\,\{I\} \quad I \wedge \neg B \implies R}{\{I\}\,\texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\,\{R\}}\,(while_2)$$

1. Prouver cette règle en utilisant le calcul de Hoare
2. Expliquer à quoi correspondent les deux obligations de preuve générées par la règle $(while_2)$

> Dans les exercices suivants, on demontrera la terminaison des while-programs avec la méthode des ensembles bien fondés, et leur correction avec le calcul de Hoare, en utilisant la règle $(while_2)$ plutôt que la règle $(while)$.

Exercice 3 (La suite Fibonacci)

Prouver la terminaison et la correction de l'algorithme suivant qui calcule le terme de rang `n` de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$.

$$\begin{cases} F_0 = F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad n \geq 2 \end{cases}$$

```
1  {n ∈ ℕ}
2  y := x; x := 1; k := 1;
3  while(k < n) do
4     t := y; y := x; x := x+t; k := k+1;
5  end
6  {x = F_n}
```

Exercice 4 (Tri à bulles)

Prouver la correction et la terminaison du while-program ci-dessous qui tri le tableau `t` de taille `n` par la méthode du tri à bulles. On introduit les prédicats :

— $sorted(t, i, j)$ défini par $\forall k.(i \leq k < j \implies t[i] \leq t[i+1])$ qui est vrai si t est trié des indices i à j.

— $bigger(t, imax, i, j)$ défini par $\forall k.(i \leq k \leq j \implies t[k] \leq t[imax])$ qui est vrai si tous les éléments de t des indices i à j sont plus petits ou égaux à $t[imax]$.

ainsi que la fonction :

— $swap(t, i, j)$ qui permute les éléments d'indice i et j dans t. En notant $t' = swap(t, i, j)$, on a : $t'[i] = t[j] \wedge t'[j] = t[i] \wedge \forall k \in [0; n) \setminus \{i, j\}.t'[k] = t[k]$.

```
1   {n ∈ ℕ ∧ n > 0}
2   i := n−1;
3   while (i > 0) do
4     j := 1;
5     while (j ≤ i) do
6       if (t[j − 1] > t[j]) then
7         t := swap(t, j − 1, j)
8       end;
9       j := j+1
10    end;
11    i := i−1
12  end
13  {sorted(t, 0, n − 1)}
```

♦

Exercice 5 (Multiplication rapide)

Prouver la terminaison et la correction de l'algorithme suivant qui calcule le produit de a par b. NB : / calcule la division entière et % le reste de la division entière.

```
1   {a ∈ ℕ ∧ b ∈ ℕ}
2   p:=0; a':=a; b':=b;
3   while (b'>0) do
4     if (b'%2 = 1) then
5       p:=p+a'
6     else
7       skip
8     endif;
9     a':=a'*2;
10    b':=b'/2
11  endwhile
12  {p = a × b}
```

♦